Analytical Differential Calculus with Integration

Han Xu
Department of Computer Science and Technology, Peking University, Beijing, China

Zhenjiang Hu
Key Laboratory of High Confidence Software Technologies (MoE),
Department of Computer Science and Technology, Peking University, Beijing, China

Abstract

Differential lambda-calculus was first introduced by Thomas Ehrhard and Laurent Regnier in 2003. Despite more than 15 years of history, little work has been done on a differential calculus with integration. In this paper, we shall propose a differential calculus with integration from a programming point of view. We show its good correspondence with mathematics, which is manifested by how we construct these reduction rules and how we preserve important mathematical theorems in our calculus. Moreover, we highlight applications of the calculus in incremental computation, automatic differentiation, and computation approximation.

2012 ACM Subject Classification Software and its engineering → General programming languages; Software and its engineering → Functional languages

Keywords and phrases Differential Calculus, Integration, Lambda Calculus, Incremental Computation, Adaptive Computing

Digital Object Identifier 10.4230/LIPIcs.ICALP.2021.143

Category Track B: Automata, Logic, Semantics, and Theory of Programming

Related Version *Full Version*: `https://arxiv.org/abs/2105.02632`

1 Introduction

Differential calculus has more than 15 years of history in computer science since the pioneer work by Thomas Ehrhard and Laurent Regnier [9]. It is, however, not well-studied from the perspective of programming languages; we would expect the profound connection of differential calculus with important fields such as incremental computation, automatic differentiation and self-adjusting computation just like how mathematical analysis connects with mathematics. We want to understand what is the semantics of the derivative of a program and how we can use these derivatives to write a program. That is, we wish to have a clear description of derivatives and introduce integration to compute from operational derivatives to the program.

The two main lines of the related work are the differential lambda-calculus [9, 8] and the change theory [7, 4, 5]. On one hand, the differential lambda-calculus uses linear substitution to represent the derivative of a term. For example, given a term $x * x$ (i.e., x^2), with the differential lambda-calculus, we may use the term $\frac{\partial x*x}{\partial x} \cdot 1$ to denote its derivative at 1. As there are two alternatives to substitute 1 for x in the term $x * x$, it gives $(1 * x) + (x * 1)$ (i.e., $2x$) as the derivative (where $+$ denotes "choice").

Despite that the differential lambda-calculus provides a concise way to analyze the alternatives of linear substitution on a lambda term, there is a gap between analysis on terms and computation on terms. For instance, let $+'$ denote our usual addition operator, and $+$ denote the choice of linear substitution. Then we have that $\frac{\partial x+'x}{\partial x} \cdot 1 = (1 +' x) + (x +' 1)$, which is far away from the expected $1 +' 1$. Moreover, it offers no method to integrate over a derivative, say $\frac{\partial t}{\partial x} \cdot y$.

On the other hand, the change theory gives a systematic way to define and propagate (transfer) changes. The main idea is to define the change of function f as *Derive* f, satisfying

$$f(x \oplus \Delta x) = f(x) \oplus (\textit{Derive } f)\ x\ \Delta x.$$

where $\oplus$ denotes an updating operation. It reads that the change over the input x by Δx results in the change over the result of $f(x)$ by (*Derive* f) x Δx. While change theory provides a general way to describe changes, the changes it described are differences (deltas) instead of derivatives. It is worth noting that derivative is not the same as delta. For example, by change theory, we can deduce that $f(x)$ will be of the form of $x * x + C$ if we know (*Derive* f) x $\Delta x = 2 * x * \Delta x + \Delta x * \Delta x$, but we cannot deduce this form if we just know that its derivative is $2 * x$, because change theory has no concept of integration or limits.

Although a bunch of work has been done on derivatives [9, 8, 7, 4, 19, 16, 21, 10, 1], there is unfortunately, as far as we are aware, little work on integration. It may be natural to ask what a derivative really means if we cannot integrate it. If there is only a mapping from a term to its derivative without its corresponding integration, how can we operate on derivatives with a clear understanding of what we actually have done?

In this paper[1], we aim at a new differential framework, having dual mapping between derivatives and integrations. With this framework, we can manifest the power of this dual mapping by proving, among others, three important theorems, namely the Newton-Leibniz formula, the Chain Rule and the Taylor's theorem.

Our key idea can be illustrated by a simple example. Suppose we have a function f mapping from an n-dimensional space to an m-dimensional space. Then, let x be $(x_1, x_2, ..., x_n)^T$, and $f(x)$ be $(f_1(x), f_2(x), ..., f_m(x))^T$. Mathematically, we can use a Jacobian matrix A to represent its derivative, which satisfies the equation

$$f(x + \Delta x) - f(x) = A\Delta x + o(\Delta x),\ \text{where } A\ = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix}$$

However, computer programs usually describe computation over data of some structure, rather than just scalar data or matrix. In this paper, we extend the idea and propose a new calculus that enables us to perform differentiation and integration on data structures. Our main contributions are summarized as follows.

- To our knowledge, we have made the first attempt of designing a calculus that provides both derivative and integral. It is an extension of the lambda-calculus with five new operators including derivatives and integrations. We give clear semantics and typing rules, and prove that it is sound and strongly normalizing. (Section 2)
- We prove three important theorems and highlight their practical application for incremental computation, automatic differentiation, and computation approximation.
  - We prove the Newton-Leibniz formula: $\int_{t_1}^{t_2} \frac{\partial t}{\partial y}|_x dx = t[t_2/y] \ominus t[t_1/y]$, which is also known as Second Fundamental Theorem of Calculus. It shows the duality between derivatives and integrations, and can be used for incremental computation. (Section 3)
  - We prove the Chain Rule: $\frac{\partial f(g\ x)}{\partial x}|_{t_1} * t = \frac{\partial f\ y}{\partial y}|_{g\ t_1} * (\frac{\partial g\ z}{\partial z}|_{t_1} * t)$. It says $\forall x, \forall x_0, (f(g(x)))' * x_0 = f'(g(x)) * g'(x) * x_0$, and can be used for incremental computation and automatic differentiation. (Section 4)

[1] A full version of this paper is available at `https://arxiv.org/abs/2105.02632`.

Terms	t	::=	c	constants of interpretable type
		|	x	variable
		|	$\lambda x : T.\, t$	lambda abstraction
		|	$t\ t$	function application
		|	$(t_1, t_2, \ldots, t_n) \mid \pi_j\ t$	n-tuple and projection
		|	$t \oplus t$	addition
		|	$t \ominus t$	subtraction
		|	$t * t$	multiplication
		|	$\frac{\partial t}{\partial x}\|_t$	derivative
		|	$\int_t^t t\ dx$	integration
		|	$inl\ t \mid inr\ t$	left/right injection
		|	$case\ t\ of\ inl\ x_1 \Rightarrow t \mid inr\ x_2 \Rightarrow t$	case analysis
		|	$fix\ t$	fix point
Types	T	::=	B	base type
		|	$(T_1, T_2, \ldots, T_n)$	product type
		|	$T \to T$	function type
		|	$T + T$	sum type
Contexts	Γ	::=	$\emptyset$	empty context
		|	$\Gamma,\ x : T$	variable binding

Figure 1 Calculus Syntax.

- We prove the Taylor's Theorem: $f\ t = \sum_{k=0}^{\infty} \frac{1}{k!}(f^{(k)}\ t_0) * (t \ominus t_0)^k$. Different from that one of the differential lambda-calculus [9], this Taylor's theorem manifests results of computation instead of analysis on occurrence of terms. It can be used for approximation of a function computation. (Section 5)

2 Calculus

In this section, we shall give a clear definition of our calculus with both derivatives and integration. We explain important insights in our design, and prove some useful properties and theorems that will be used later.

2.1 Syntax

Our calculus, as defined in Figure 1, is an extension of the simply-typed lambda calculus [20]. Besides the usual constant, variable, lambda abstraction, function application, and tuple, it introduces five new operations: addition $\oplus$, subtraction $\ominus$, multiplication $*$, derivative $\frac{\partial t}{\partial x}|_t$ and integration $\int_t^t t\ dx$. The three binary operations, namely $\oplus$, $\ominus$, and $*$, are generalizations of those from our mathematics. Intuitively, $x \oplus \Delta$ is for updating x with change Δ, $\ominus$ for canceling updates, and * for distributing updates. We build up terms from terms of base types (such as $\mathbb{R}$, $\mathbb{C}$), and on each base type we require these operations satisfy the following properties:

$$\frac{c : T \in \Gamma}{\Gamma \vdash c : T}\ (\textsc{TCon}) \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ (\textsc{TVar})$$

$$\frac{\Gamma \vdash t : T_1}{\Gamma \vdash inl\ t : T_1 + T_2}\ (\textsc{TInl}) \qquad \frac{\Gamma \vdash t : T_2}{\Gamma \vdash inr\ t : T_1 + T_2}\ (\textsc{TInr})$$

$$\frac{\Gamma \vdash t_1 : T^* \quad \Gamma \vdash t_2 : T^*}{\Gamma \vdash t_1 \oplus t_2 : T^*}\ (\textsc{TAdd}) \qquad \frac{\Gamma \vdash t_1 : T^* \quad \Gamma \vdash t_2 : T^*}{\Gamma \vdash t_1 \ominus t_2 : T^*}\ (\textsc{TSub})$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1 . t : T_1 \to T_2}\ (\textsc{TAbs}) \qquad \frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}\ (\textsc{TApp})$$

$$\frac{\Gamma \vdash t : T \to T}{\Gamma \vdash fix\ t : T}\ (\textsc{TFix}) \qquad \frac{\Gamma \vdash t_1 : T_1 \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \frac{\partial t_2}{\partial x}|_{t_1} : \frac{\partial T_2}{\partial T_1}}\ (\textsc{TDer})$$

$$\frac{\forall j \in [1,n], \Gamma \vdash t_j : T_j}{\Gamma \vdash (t_1, t_2, ..., t_n) : (T_1, T_2, ..., T_n)}\ (\textsc{TPair}) \qquad \frac{\forall j \in [1,n], \Gamma \vdash t : (T_1, T_2, ..., T_n)}{\Gamma \vdash \pi_j\ t : T_j}\ (\textsc{TProj})$$

$$\frac{\Gamma \vdash t_1 : \frac{\partial T^*}{\partial T} \quad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1 * t_2 : T^*}\ (\textsc{TMul})$$

$$\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T \quad \Gamma, x : T \vdash t : \frac{\partial T^*}{\partial T}}{\Gamma \vdash \int_{t_1}^{t_2} t\ dx : T^*}\ (\textsc{TInt})$$

$$\frac{\Gamma, x_1 : T_1 \vdash t_1 : T \quad \Gamma, x_2 : T_2 \vdash t_2 : T \quad \Gamma \vdash t : T_1 + T_2}{\Gamma \vdash case\ t\ of\ inl\ x_1 \Rightarrow t_1 \mid inr\ x_2 \Rightarrow t_2 : T}\ (\textsc{TCase})$$

Figure 2 Typing Rules.

- The addition and multiplication are associative and commutative, i.e., $(a \oplus b) \oplus c = a \oplus (b \oplus c)$, $a \oplus b = b \oplus a$, $(a * b) * c = a * (b * c)$, $a * b = b * a$.
- The addition and the subtraction are cancellable, i.e., $(a \oplus b) \ominus b = a$ and $(a \ominus b) \oplus b = a$.
- The multiplication is distributive over addition, i.e., $a * (b \oplus c) = a * b \oplus a * c$.

▶ **Example 1** (Basic Operations on Real Numbers)**.** For real numbers $r_1, r_2 \in \mathbb{R}$, we have the following definitions.

$$\begin{array}{lcl} r_1 \oplus r_2 & = & r_1 + r_2 \\ r_1 \ominus r_2 & = & r_1 - r_2 \\ r_1 * r_2 & = & r_1\ r_2 \end{array}$$

We use $\frac{\partial t_1}{\partial x}|_{t_2}$ to denote derivative of t_1 over x at point t_2, and $\int_{t_1}^{t_2} t\ dx$ to denote integration of t over x from t_1 to t_2.

2.2 Typing

As defined in Figure 1, we have base types (denoted by B), tuple types, function types, and sum type. To make our later typing rules easy to understand, we introduce the following type notations.

$$\begin{array}{llcll} \text{Type} & T^* & ::= & B & \text{base type} \\ & & \mid & (T^*, T^*, ..., T^*) & \text{product type} \\ & & \mid & T \to T^* & \text{arrow type} \end{array}$$

T^* means the types that are addable (i.e., updatable through $\oplus$). We view the addition between functions, tuples and base type terms as valid, which will be showed by our reduction rules later. But here, we forbid the addition and subtraction between sum types because we view updates such as $inl\ 0 \oplus inr\ 1$ as invalid. If we want to update the change to a term of sum types anyway, we may do case analysis such as $case\ t\ of\ inl\ x_1 \Rightarrow inl\ (x_1 \oplus ...) \mid inr\ x_2 \Rightarrow (x_2 \oplus ...)$.

Next, we introduce two notations for derivatives on types:

$$\frac{\partial T}{\partial \mathsf{B}} = T,$$

$$\frac{\partial T}{\partial (T_1, T_2, ..., T_n)} = (\frac{\partial T}{\partial T_1}, \frac{\partial T}{\partial T_2}, ..., \frac{\partial T}{\partial T_n}).$$

The first notation says that with the assumption that differences (subtraction) of values of base types are of base types, the derivative over base types has no effect on the result type. And, the second notation resembles partial differentiation. Note that we do not consider derivatives on functions because even for functions on real numbers, there is no good mathematical definition for them yet. Therefore, we do not have a type notation for $\frac{\partial T}{\partial (T_1 \to T_2)}$. Besides, because we forbid the addition and subtraction between the sum types, we will iew the differentiation of the sum types as invalid, so we do not have notations for $\frac{\partial T}{\partial (T_1 + T_2)}$ either.

Figure 2 shows the typing rules for the calculus. The typing rules for constant, variable, lambda abstraction, function application, tuple, and projection are nothing special. The typing rules for addition and subtraction are natural, but the rest three kinds of rules are more interesting. Rule TMul the typing rule for $t_1 * t_2$. If t_1 is a derivative of T_1 over T_2, and t_2 is of type T_2, then multiplication will produce a term of type T_1. This may be informally understood from our familiar equation $\frac{\triangle Y}{\triangle X} * \triangle X = \triangle Y$. Rule TDer shows introduction of the derivative type through a derivative operation, while Rule TInt cancellation of the derivative type through an integration operation.

2.3 Semantics

We will give a two-stage semantics for the calculus. At the first stage, we assume that all the constants (values and functions) over the base types are *interpretable* in the sense there is a default well-defined interpreter to evaluate them. At the second stage, the important part of this paper, we define a set of reduction rules and use the full reduction strategy to compute their normal form, which enjoys good properties of soundness, confluence, and strong normalization.

More specifically, after the full reduction of a term in our calculus, every subterm (now in a normal form of interpretable types) outside the lambda function body will be interpretable on base types, which will be proved in the full version. In other words, our calculus helps to reduce a term to a normal form which is interpretable on base types, and leave the remaining evaluations to interpretation on base types. We will not give reduction rules to the operations on base types because we do not want to touch on implementations of primitive functions on base types.

For simplicity, in this paper we will assume that the important properties such as the Newton-Leibniz formula, the Chain Rule, and the Taylor's theorem, are satisfied by all the primitive functions and their closures through addition, subtraction, multiplication, derivative and integration. This assumption may seem too strong, since not all primitive functions on base types meet this assumption. However, it would make sense to start with the primitive functions meeting these requirements to build our system, and extend it later with other primitive functions.

$$\frac{t_0 : \mathsf{B}}{\frac{\partial(t_1,t_2,...,t_n)}{\partial x}|_{t_0} \rightarrow (\frac{\partial t_1}{\partial x}|_{t_0}, \frac{\partial t_2}{\partial x}|_{t_0}, ..., \frac{\partial t_n}{\partial x}|_{t_0})} \quad \text{(EAppDer1)}$$

$$\frac{t_0 : \mathsf{B}}{\frac{\partial inl/inr\ t}{\partial x}|_{t_0} \rightarrow inl/inr\ \frac{\partial t}{\partial x}|_{t_0}} \quad \text{(EAppDer2)}$$

$$\frac{t_0 : \mathsf{B}}{\frac{\partial(\lambda y{:}T.t)}{\partial x}|_{t_0} \rightarrow \lambda y : T.\frac{\partial t}{\partial x}|_{t_0}} \quad \text{(EAppDer3)}$$

$$\frac{\forall i \in [1,n],\ t_{i*} = (t_1, t_2 ..., t_{i-1}, x_i, t_{i+1} ..., t_n)}{\frac{\partial t}{\partial x}|_{(t_1,t_2,...,t_n)} \rightarrow (\frac{\partial t[t_{1*}/x]}{\partial x_1}|_{t_1}, \frac{\partial t[t_{2*}/x]}{\partial x_2}|_{t_2}, ..., \frac{\partial t[t_{n*}/x]}{\partial x_n}|_{t_n})} \quad \text{(EAppDer4)}$$

$$\frac{t_1, t_2 : \mathsf{B}}{\int_{t_1}^{t_2}(t_{11}, t_{12}, ...t_{1n})dx \rightarrow (\int_{t_1}^{t_2} t_{11}dx, \int_{t_1}^{t_2} t_{12}dx, ..., \int_{t_1}^{t_2} t_{1n}dx)} \quad \text{(EAppInt1)}$$

$$\frac{t_1, t_2 : \mathsf{B}}{\int_{t_1}^{t_2} inl/inr\ t\ dx \rightarrow inl/inr\ \int_{t_1}^{t_2} t\ dx} \quad \text{(EAppInt2)}$$

$$\frac{t_1, t_2 : \mathsf{B}}{\int_{t_1}^{t_2} \lambda y : T_2.tdx \rightarrow \lambda y : T_2.\int_{t_1}^{t_2} tdx} \quad \text{(EAppInt3)}$$

$$\frac{\forall i \in [1,n], t_{i*} = (t_{21} ..., t_{2i-1}, x_i, t_{1i+1} ..., t_{1n})}{\int_{(t_{11},t_{12},...t_{1n})}^{(t_{21},t_{22},...,t_{2n})} tdx \rightarrow \int_{t_{11}}^{t_{21}} \pi_1(t[t_{1*}/x])dx_1 \oplus ... \oplus \int_{t_{1n}}^{t_{2n}} \pi_n(t[t_{n*}/x])dx_n} \quad \text{(EAppInt4)}$$

Figure 3 Reduction Rules for Derivative and Integration.

2.4 Reduction Rules

Our calculus is an extension of simply-typed lambda-calculus. Our lambda abstraction and application are nothing different from the simply-typed lambda calculus, and we have the reduction rule:

$$(\lambda x : T.\, t)t_1 \rightarrow t[t_1/x].$$

We use an n-tuple to model structured data and projection π_j to extract j-th component from a tuple, and we have the following reduction rule:

$$\pi_j(t_1, t_2, ...t_n) \rightarrow t_j.$$

Similarly, we have reduction rules for the case analysis:

$$case\ (inl\ t)\ of\ inl\ x_1 \Rightarrow t_1 \mid inr\ x_2 \Rightarrow t_2 \rightarrow t_1[t/x_1]$$

$$case\ (inr\ t)\ of\ inl\ x_1 \Rightarrow t_1 \mid inr\ x_2 \Rightarrow t_2 \rightarrow t_2[t/x_2]$$

Besides, we introduce fix-point operator to deal with recursion:

$$fix\ f \rightarrow f\ (fix\ f)$$

It is worth noting that tuples, having a good correspondence in mathematics, should be understood as structured data instead of high-dimensional vectors because there are some operations that are different from those in mathematics. As will be seen later, there is difference between our multiplication and matrix multiplication, and derivative and integration on tuples of tuples has no correspondence to mathematical objects.

The core reduction rules in our calculus are summarized in Figure 3, which define three basic cases for both reducing derivative terms and integration terms. For derivative, we use $\frac{\partial t}{\partial x}|_{t_0}$ to denote the derivative of t over x at point t_0, and we have four reduction rules:

- Rule EAppDer1 is to distribute point $t_0 : \mathsf{B}$ into a tuple. This resembles the case in mathematics; if we have a function f defined by $f(x) = (f_1(x), f_2(x), \ldots, f_m(x))^T$, its derivative will be $(\frac{\partial f_1}{\partial x}, \frac{\partial f_2}{\partial x}, \ldots, \frac{\partial f_m}{\partial x})^T$. For example, if we have a function $f : \mathbb{R} \to (\mathbb{R}, \mathbb{R})$ defined by $f(x) = (x, x * x)$, then its derivative will be $(1, 2 * x)$.
- Rule EAppDer2 is similar to Rule EAppDer1.
- Rule EAppDer3 is to distribute point $t_0 : \mathsf{B}$ into a lambda abstraction. Again this is very natural in mathematics. For example, for function $f(x) = \lambda y : B.\, x * y$, then we would have its derivative on x as $\lambda y : B.y$.
- Rule EAppDer4 is to deal with partial differentiation, similar to the Jacobian matrix in mathematics (as shown in the introduction). For example, if we have a function that maps a pair (x, y) to $(x * x, x * y \oplus y)$, which may be written as $\lambda z : (\mathsf{B}, \mathsf{B}).\, (\pi_1 z * \pi_1 z, (\pi_1 z * \pi_2 z \oplus \pi_2 z))$ then we would have its derivative $\frac{\partial (f\ z)}{\partial z}|_{(x,y)}$ as $((2 * x, y), (0, x \oplus 1))$.

Similarly, we can define four reduction rules for integration. Rules EAppInt1,EAppInt2 and EAppInt3 are simple. Rule EAppInt4 is worth more explanation. It is designed to establish the Newton-Leibniz formula

$$\int_{t_1}^{t_2} \frac{\partial t}{\partial y}|_x dx = t[t_2/y] \ominus t[t_1/y]$$

when t_1 and t_2 are tuples:

$$\int_{(t_{11}, t_{12}, \ldots, t_{1n})}^{(t_{21}, t_{22}, \ldots, t_{2n})} \frac{\partial t}{\partial y}|_x dx = t[(t_{21}, t_{22}, ..., t_{2n})/y] \ominus t[(t_{11}, t_{12}, ..., t_{1n})/y].$$

So we design the rule to have

$$\int_{t_{1j}}^{t_{2j}} \frac{\partial t[(t_{21}, ..., t_{2(j-1)}, x'_j, t_{1(j+1)}, ..., t_{1n})/y]}{\partial x'_j}|_{x_j} dx_j = \int_{(t_{21}, \ldots, t_{2(j-1)}, t_{1j}, t_{1(j+1)}, \ldots, t_{1n})}^{(t_{21}, \ldots, t_{2(j-1)}, t_{2j}, t_{1(j+1)}, \ldots, t_{1n})} \frac{\partial t}{\partial y}|_x dx.$$

Notice that under our evaluation rules on derivative, $\pi_j(\frac{\partial t}{\partial x}|_{x=(x_1, x_2, \ldots, x_n)})$ will be equal to the derivative of t to its j-th parameter x_j, so the integration will lead us to the original t.

Finally, we discuss the reduction rules for the three new binary operations, as summarized in Figure 4. The addition $\oplus$ is introduced to support the reduction rule of integration. It is also useful in proving the theorem and constructing the formula. We can understand the two reduction rules for addition as the addition of high-dimension vectors and functions respectively. Similarly, we can have two reduction rules for subtraction $\ominus$. The operator $*$ was introduced as a powerful tool for constructing the Chain Rule and the Taylor's theorem. The first two reduction rules can be understood as multiplications of a scalar with a function and a high-dimension vector respectively, while the last one can be understood as the multiplication on matrix. For example, we have

$$((1, 4), (2, 5), (3, 6)) * (7, 8, 9) = (50, 122)$$

which corresponds to the following matrix multiplication.

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 7 \\ 8 \\ 9 \end{pmatrix} = \begin{pmatrix} 50 \\ 122 \end{pmatrix}$$

$$(t_{11},...,t_{1n}) \oplus (t_{21},...,t_{2n}) \ \to \ (t_{11} \oplus t_{21},...,t_{1n} \oplus t_{2n}) \quad \text{(EAppAdd1)}$$

$$(\lambda x : T.\, t_1) \oplus (\lambda y : T.\, t_2) \ \to \ \lambda x : T.\, (t_1 \oplus t_2[y/x]) \quad \text{(EAppAdd2)}$$

$$(t_{11},...,t_{1n}) \ominus (t_{21},...,t_{2n}) \ \to \ (t_{11} \ominus t_{21},...,t_{1n} \ominus t_{2n}) \quad \text{(EAppSub1)}$$

$$(\lambda x : T.\, t_1) \ominus (\lambda y : T.\, t_2) \ \to \ \lambda x : T.\, (t_1 \ominus t_2[y/x]) \quad \text{(EAppSub2)}$$

$$\frac{t_0 : \mathsf{B}}{(t_1, t_2, ..., t_n) * t_0 \ \to \ (t_1 * t_0, t_2 * t_0, ..., t_n * t_0)} \quad \text{(EAppMul1)}$$

$$\frac{t_0 : \mathsf{B}}{(\lambda x : T.t) * t_0 \ \to \ \lambda x : T.(t * t_0)} \quad \text{(EAppMul2)}$$

$$\frac{t_0 : \mathsf{B}}{(inl/inr\ t) * t_0 \ \to \ inl/inr\ (t * t_0)} \quad \text{(EAppMul3)}$$

$$\frac{t_1 : (t_{11}, t_{12}, ...t_{1n}), t_2 : (t_{21}, t_{22}, ...t_{2n})}{t_1 * t_2 \ \to \ (t_{11} * t_{21}) \oplus (t_{12} * t_{22}) \oplus ... \oplus (t_{1n} * t_{2n})} \quad \text{(EAppMul4)}$$

Figure 4 Reduction Rules for Addition, Subtraction and Multiplication.

It is worth noting that while they are similar, $*$ is different from the matrix multiplication operation. For example, we cannot write x as an m-dimensional vector (or $m * 1$ matrix) in Taylor's theorem because no matrix A is well-performed under $A * x * x$, but we can write Taylor's Theorem easily under our framework. In the matrix representation, the number of rows of the first matrix and the number of columns of the second matrix must be equal so that we can perform multiplication on them. This means, we can only write case $m = 1$'s Taylor's theorem in matrices, while our version can perform for any tuples.

2.5 Properties

Next, we prove some properties of our calculus. The proof is rather routine with some small variations.

▶ **Lemma 2** (Properties). This calculus has the properties of progress, preservation and confluence. Moreover, if a term t does not contain subterms $fix\ t'$, then t is strong normalizable.

Proof. Full proof is in the full version, which is adapted from the standard proof. ◀

2.6 Term Equality

We need to talk a bit more on equality because we do not consider reduction or calculation on primitive functions. This notion of equality has little to do with our evaluation but has a lot to do with the equality of primitive functions. Using this notion of equality, we can compute the result from completely different calculations. This will be used in our later proof of the three theorems.

Since we have proved the confluence property, we know that every term has at most one normal form after reduction. Thus, we can define our equality based on their normal forms; the equality between unnormalizable terms is undefined.

▶ **Definition 3** (Term Equality). An open term t_1 is said to be equal to a term t_2, if and only if for all free variables $x_1, x_2, ..., x_n$ in t_1 and t_2, for all closed and weak-normalizable term u_i whose type is the same as that of x_i, we have $t_1[u_1/x_1, ..., u_n/x_n] = t_2[u_1/x_1, ..., u_n/x_n]$.

A closed-term $t_1 = t_2$, if their normal forms n_1 and n_2 have the relation that $n_1 = n_2$, where a normal form n_1 is said to be equal to another normal form n_2, if they satisfy one of the following rules:

- (1) n_1 is a of type iB (Type iB is used to capture terms with base type constants and functions, being defined by $iB = iB \to iB \mid B$). A normal form of type iB is interpretable by the base type interpreter. Detailed proof is in the full version), then n_2 has to be of the same type, and under the base type interpretation, n_1 is equal to n_2;
- (2) n_1 is $(t_1, t_2, ..., t_n)$, then n_2 has to be $(t'_1, t'_2, ..., t'_n)$, and $\forall j \in [1, n], t_j$ is equal to t'_j;
- (3) n_1 is $\lambda x : T.t$, then n_2 has to be $\lambda y : T.t'$ (y can be x), and $n_1\ x$ is equal to $n_2\ x$.
- (4) n_1 is $inl\ t'_1$, then n_2 has to be $inl\ t'_2$, and t'_1 is equal to t'_2.
- (5) n_1 is $inr\ t'_1$, then n_2 has to be $inr\ t'_2$, and t'_1 is equal to t'_2.

▶ **Lemma 4.** The equality is reflexive, transitive and symmetric for weak-normalizable terms.

Proof. Based on the equality of terms of base types, we can prove it by induction. ◀

▶ **Lemma 5.** The equality is consistent, e.g., we can not prove equality between arbitrary two terms.

Proof. Notice that except for the equality introduced by the base type interpreter, other equality inferences all preserve the type. So for arbitrary t_1 of type (B, B) and t_2 of type B, we can not prove equality between them. ◀

Next we give some lemmas that will be used later in our proof. It is relatively unimportant to the mainline of our calculus, so we put their proofs in the full version.

▶ **Lemma 6.** If $t_1 \rho^* t'_1, t_2 \rho^* t'_2$, then $t_1[t_2/x]\rho^* t'_1[t'_2/x]$.

▶ **Lemma 7.** If $t_1 = t'_1$, $t_2 = t'_2$, then $t_1 \oplus t_2 = t'_1 \oplus t'_2$.

▶ **Lemma 8.** For a term t, for any subterm s, if the term s'=s, then t[s'/s]=t. (We only substitute the subterm s, but not other subterms same as s)

▶ **Lemma 9.** If $t_1 * (t_2 \oplus t_3)$ and $(t_1 * t_2) \oplus (t_1 * t_3)$ are weak-normalizable, then $t_1 * (t_2 \oplus t_3) = (t_1 * t_2) \oplus (t_1 * t_3)$.

▶ **Lemma 10.** If $(t_1 \ominus t_2) \oplus (t_2 \ominus t_3)$ and $t_1 \ominus t_3$ are weak-normalizable, then $(t_1 \ominus t_2) \oplus (t_2 \ominus t_3) = t_1 \ominus t_3$.

3 Newton-Leibniz's Formula

The first important theorem we will give is the Newton-Leibniz's formula, which ensures the duality between derivatives and integration. This theorem lays a solid basis for our calculus.

▶ **Theorem 11** (Newton-Leibniz). Let t contain no free occurrence of x, and both $\int_{t_1}^{t_2} \frac{\partial t}{\partial y}|_x dx$ and $t[t_2/y] \ominus t[t_1/y]$ are well-typed and weak-normalizable. Then we have

$$\int_{t_1}^{t_2} \frac{\partial t}{\partial y}|_x dx = t[t_2/y] \ominus t[t_1/y].$$

Proof. If t_1, t_2 or t is not closed, then we need to prove $\forall u_1, ..., u_n$, we have

$$(\int_{t_1}^{t_2} \frac{\partial t}{\partial y}|_x dx)[u_1/x_1, ..., u_n/x_n] = (t[t_2/y] \ominus t[t_1/y])[u_1/x_1, ..., u_n/x_n].$$

By freezing $u_1, ..., u_n$, we can apply the substitution $[u_1/x_1, ..., u_n/x_n]$ to make every term closed. So, for simplicity, we will assume t, t_1 and t_2 to be closed.

We prove this by induction on types.

- Case: t_1,t_2 and t are of base types. By the confluence lemma, we know there exists the normal form t', t'_1 and t'_2 of the term t, t_1 and t_2. Also, we know $\int_{t_1}^{t_2} \frac{\partial t}{\partial y}|_x dx = \int_{t'_1}^{t'_2} \frac{\partial t'}{\partial y}|_x dx$ and $t[t_2/y] \ominus t[t_1/y] = t'[t'_2/y] \ominus t'[t'_1/y]$. Since on base types we have $\int_{t'_1}^{t'_2} \frac{\partial t'}{\partial y}|_x dx = t'[t'_2/y] \ominus t'[t'_1/y]$, we have $\int_{t_1}^{t_2} \frac{\partial t}{\partial y}|_x dx = t[t_2/y] \ominus t[t_1/y]$.
- Case: t_1,t_2 are of base types, t is of type $(T_1, T_2, ..., T_n)$. By the confluence lemmas, there exist a normal form $(t'_{11}, t'_{12}, ..., t'_{1n})$ for t. Using Rules (EAppInt1) and (EAppDer1), we know

$$\begin{array}{rcl} \int_{t_1}^{t_2} \frac{\partial t}{\partial y}|_x dx & = & \int_{t_1}^{t_2} \frac{\partial (t'_{11}, t'_{12}, ..., t'_{1n})}{\partial y}|_x dx \\ & = & \int_{t_1}^{t_2} (\frac{\partial t'_{11}}{\partial y}|_x, \frac{\partial t'_{12}}{\partial y}|_x, ..., \frac{\partial t'_{1n}}{\partial y}|_x) dx \\ & = & (\int_{t_1}^{t_2} \frac{\partial t'_{11}}{\partial y}|_x dx, \int_{t_1}^{t_2} \frac{\partial t'_{12}}{\partial y}|_x dx, ..., \int_{t_1}^{t_2} \frac{\partial t'_{1n}}{\partial y}|_x dx) \end{array}$$

  On the other hand, we have

$$\begin{array}{l} t[t_2/y] \ominus t[t_1/y] \\ \quad = (t'_{11}[t_2/y], t'_{12}[t_2/y], ..., t'_{1n}[t_2/y]) \ominus (t'_{11}[t_1/y], t'_{12}[t_1/y], ..., t'_{1n}[t_1/y]) \\ \quad = (t'_{11}[t_2/y] \ominus t'_{11}[t_1/y], t'_{12}[t_2/y] \ominus t'_{12}[t_1/y], ..., t'_{1n}[t_2/y] \ominus t'_{1n}[t_1/y]) \end{array}$$

  By induction, we have $\forall j \in [1, n], \int_{t_1}^{t_2} \frac{\partial t'_{1j}}{\partial y}|_x dx = t'_{1j}[t_2/y] \ominus t'_{1j}[t_1/y]$, so we have proven the case.
- Case: t_1,t_2 are of base types, t is of type $A \to B$. By Lemma 8, we can use $\lambda z : A.t\ z$ (for simiplicity, we use $\lambda z : A.t'$ where $t' = t\ z$) to substitute for t, where z is a fresh variable. Now, we have for any u,

$$\begin{array}{rcl} (\int_{t_1}^{t_2} \frac{\partial t}{\partial y}|_x dx)\ u & = & (\int_{t_1}^{t_2} \frac{\partial \lambda z:A.t'}{\partial y}|_x dx)\ u \\ & = & \lambda z : A.(\int_{t_1}^{t_2} \frac{\partial t'}{\partial y}|_x dx)\ u \\ & = & \int_{t_1}^{t_2} \frac{\partial t'[u/z]}{\partial y}|_x dx \end{array}$$

  and on the other hand, since z is free in t_1 and t_2, we have

$$\begin{array}{rcl} (t[t_2/y] \ominus t[t_1/y])\ u & = & ((\lambda z : A.t')[t_1/y] \ominus (\lambda z : A.t')[t_2/y])\ u \\ & = & \lambda z : A.(t'[t_2/y] \ominus t'[t_1/y])\ u \\ & = & (t'[t_2/y] \ominus t'[t_1/y])[u/z] \\ & = & (t'[u/z])[t_2/y] \ominus (t'[u/z])[t_1/y] \end{array}$$

  By induction (on B), we know $\int_{t_1}^{t_2} \frac{\partial t'[u/z]}{\partial y}|_x dx = (t'[u/z])[t_2/y] \ominus (t'[u/z])[t_1/y]$, thus we have proven the case.
- Case: t_1,t_2 are of base types, t is of type $T_1 + T_2$. This case is impossible because the righthand term is not well-typed.

- Case: t_1,t_2 are of type $(T_1, T_2, ..., T_n)$, t is of any type T. By using the confluence lemma, we know there exist the normal forms $(t'_{11}, t'_{12}, ..., t'_{1n})$ and $(t'_{21}, t'_{22}, ..., t'_{2n})$ for t_1 and t_2 respectively.
  Applying Rules (EAppDer3) and (EAppInt3), we have

$$\begin{aligned}\int_{t_1}^{t_2} \tfrac{\partial t}{\partial y}|_x dx &= \int_{(t'_{11},t'_{12},...,t'_{1n})}^{(t'_{21},t'_{22},...,t'_{2n})} \tfrac{\partial t}{\partial y}|_x dx \\ &= \int_{t'_{11}}^{t'_{21}} \pi_1(\tfrac{\partial t}{\partial y}|_x[(x_1, t'_{12}, ...t'_{1n})/x])dx_1 \oplus \cdots \oplus \\ &\quad \int_{t'_{1n}}^{t'_{2n}} \pi_n(\tfrac{\partial t}{\partial y}|_x[(t'_{21}, t'_{22}, ..., x_n)/x])dx_n\end{aligned}$$

  Notice that there is no occurrence of x in t, so we have

$$\begin{aligned}&\int_{t'_{1j}}^{t'_{2j}} \pi_j(\tfrac{\partial t}{\partial y}|_x[(t'_{21}, t'_{22}, ..., t'_{2(j-1)}, x_j, t'_{1(j+1)}, ..., t'_{1n})/x])dx_j \\ &= \int_{t'_{1j}}^{t'_{2j}} \pi_j(\tfrac{\partial t}{\partial y}|_{(t'_{21},t'_{22},...,t'_{2(j-1)},x_j,t'_{1(j+1)},...,t'_{1n})})dx_j \\ &= \int_{t'_{1j}}^{t'_{2j}} \pi_j(\tfrac{\partial t[t_{1*}/y]}{\partial x_1}|_{t'_{21}}, \tfrac{\partial t[t_{2*}/y]}{\partial x_2}|_{t'_{22}}, ..., \tfrac{\partial t[t_{(j-1)*}/y]}{\partial x_{j-1}}|_{t'_{2(j-1)}}, \\ &\qquad \tfrac{\partial t[t_{j*}/y]}{\partial x'_j}|_{x_j}, \tfrac{\partial t[t_{(j+1)*}/y]}{\partial x_{j+1}}|_{t'_{1(j+1)}}, ..., \tfrac{\partial t[t_{n*}/y]}{\partial x_n}|_{t'_{1n}})dx_j \\ &= \int_{t'_{1j}}^{t'_{2j}} \tfrac{\partial t[(t'_{21},t'_{22},...,t'_{2(j-1)},x'_j,t'_{1(j+1)},...,t'_{1n})/y]}{\partial x'_j}|_{x_j} dx_j\end{aligned}$$

  By induction (on the case where t_1, t_2 are of type T_j, t is of type T), we have

$$\begin{aligned}&\int_{t'_{1j}}^{t'_{2j}} \tfrac{\partial t[(t'_{21},t'_{22},...,t'_{2(j-1)},x'_j,t'_{1(j+1)},...,t'_{1n})/y]}{\partial x'_j}|_{x_j} dx_j \\ &= (t[(t'_{21}, t'_{22}, ..., t'_{2(j-1)}, x'_j, t'_{1(j+1)}, ..., t'_{1n})/y])[t'_{2j}/x'_j] \ominus \\ &\qquad (t[(t'_{21}, t'_{22}, ..., t'_{2(j-1)}, x'_j, t'_{1(j+1)}, ..., t'_{1n})/y])[t'_{1j}/x'_j] \\ &= (t[(t'_{21}, t'_{22}, ..., t'_{2(j-1)}, t'_{2j}, t'_{1(j+1)}, ..., t'_{1n})/y]) \ominus \\ &\qquad (t[(t'_{21}, t'_{22}, ..., t'_{2(j-1)}, t'_{1j}, t'_{1(j+1)}, ..., t'_{1n})/y])\end{aligned}$$

  Note that the last equation holds because x'_j is a fresh variable and t has no occurrence of x'_j.
  Now we have the following calculation.

$$\begin{aligned}&\int_{t_1}^{t_2} \tfrac{\partial t}{\partial y}|_x dx \\ =\ & \{ \text{ all the above } \} \\ & ((t[(t'_{21}, t'_{12}, ..., t'_{1n})/y]) \ominus (t[(t'_{11}, t'_{12}, ..., t'_{1n})/y])) \oplus \\ & \quad ((t[(t'_{21}, t'_{22}, ..., t'_{1n})/y]) \ominus (t[(t'_{21}, t'_{12}, ..., t'_{1n})/y])) \oplus \cdots \oplus \\ & \quad\quad ((t[(t'_{21}, t'_{22}, ..., t'_{2n})/y]) \ominus (t[(t'_{21}, t'_{22}, ..., t'_{1n})/y])) \\ =\ & \{ \text{ Lemma 10 } \} \\ & (t[(t'_{21}, t'_{22}, ..., t'_{2n})/y]) \ominus (t[(t'_{11}, t'_{12}, ..., t'_{1n})/y]) \\ =\ & \{ \text{ Lemma 6 } \} \\ & t[t_2/y] \ominus t[t_1/y]\end{aligned}$$

  Thus we have proven the theorem. ◀

Application: Incremental Computation

A direct application is incrementalization [17, 7, 11]. Given a function $f(x)$, if the input x is changed by Δ, then we can obtain its incremental version of $f(x)$ by $f'(x, \Delta)$.

$$f(x \oplus \Delta) = f(x) \oplus f'(x, \Delta)$$

where f' satisfies that

$$f'(x,\Delta) = \int_x^{x\oplus\Delta} \frac{\partial f(x)}{\partial x}|_x \; dx.$$

▶ **Example 12** (Averaging a Pair of Real numbers)**.** As a simple example, consider the average of a pair of real numbers

$$\begin{array}{lcl} average & :: & (\mathbb{R},\mathbb{R}) \to \mathbb{R} \\ average & = & \lambda x.(\pi_1(x)+\pi_2(x))/2 \end{array}$$

Suppose that we want to get an incremental computation of *average* at $x = (x_1, x_2)$ when the first element x_1 is changed to $x_1 + d$ while the second component x_2 is kept the same. The incremental computation is defined by

$$inc(x,d) = average(x,(d,0)) = \int_x^{x\oplus(d,0)} \frac{\partial average(x)}{\partial x}|_x \; dx = \frac{d}{2}$$

which is efficient.

4 Chain Rule

The Chain Rule is another important theorem of the relation between function composition and derivatives. This Chain Rule in our calculus has many important applications in automatic differentiation and incremental computation.

▶ **Theorem 13** (Chain Rule)**.** Let $f : T_1 \to T$, $g : T_2 \to T_1$. If both $\frac{\partial f(g\ x)}{\partial x}|_{t_1} * t$ and $\frac{\partial f\ y}{\partial y}|_{(g\ t_1)} * (\frac{\partial g\ z}{\partial z}|_{t_1} * t)$ are well-typed and weak-normalizable. Then for any $t, t_1 : T_2$, we have

$$\frac{\partial f(g\ x)}{\partial x}|_{t_1} * t = \frac{\partial f\ y}{\partial y}|_{(g\ t_1)} * (\frac{\partial g\ z}{\partial z}|_{t_1} * t).$$

Proof. Like in the proof of Theorem 11, for simplicity, we assume that f, g, t and t_1 are closed. Furthermore, we assume that t and t_1 are in normal form. We prove this by induction on types.

- Case T, T_2 are base types, and T_1 is any type. To be well-typed, T_1 must contain no $\to$ or $+$ type. So for simplicity, we suppose T_1 to be $(B, B, B, ..., B)$ of n-tuples, but the technique below can be applied to any T_1 type (such as tuples of tuples) that makes the term well-typed.
  First we notice that

$$\begin{array}{lcl} g\ z & = & (\pi_1(g\ z), \pi_2(g\ z), ..., \pi_n(g\ z)) \\ & = & ((\lambda b' : B.\pi_1(g\ b'))\ z, (\lambda b' : B.\pi_2(g\ b'))\ z, ..., (\lambda b' : B.\pi_n(g\ b'))\ z) \end{array}$$

  and for any j, we notice that $\pi_j(g\ b')$ has only one free variable of base type, so it can be reduced to a normal form, say E_j, of base type. Let g_j be $\lambda b' : B.E_j$, then we have $g\ z = (g_1\ z, g_2\ z, ..., g_n\ z)$.
  Next, we deal with the term f:

$$\begin{array}{lcl} f & = & \lambda a : T_1.(f\ a) \\ & = & \lambda a : T_1.((\lambda y_1 : B.\, \lambda y_2 : B., ... \lambda y_n : B.\, (f\ (y_1, y_2, ..., y_n)))\ \pi_1(a)\ \pi_2(a) ...\ \pi_n(a)) \end{array}$$

and we know that $(f\ (y_1, y_2, ..., y_n))$ only contains base type free variables, so it can be reduced to a base type normal form, say N, so we have

$$f = \lambda a : T.\ ((\lambda y_1 : B.\lambda y_2 : B., ...\lambda y_n : B.\ N)\ \pi_1(a)\ \pi_2(a)...\ \pi_n(a)).$$

Now, we can calculate as follows:

$$\begin{aligned}
&\tfrac{\partial f\ (g\ x)}{\partial x}|_{t_1} * t \\
&\quad = \tfrac{\partial (\lambda a{:}T.(\lambda y_1{:}B.\lambda y_2{:}B.,...\lambda y_n{:}B.N)\ \pi_1(a)\ \pi_2(a)...\ \pi_n(a))\ (g_1\ x, g_2\ x, ..., g_n\ x)}{\partial x}|_{t_1} * t \\
&\quad = \tfrac{\partial (\lambda y_1{:}B.\lambda y_2{:}B.,...\lambda y_n{:}B.N)\ (g_1\ x)\ (g_2\ x)...\ (g_n\ x)}{\partial x}|_{t_1} * t \\
&\quad = \tfrac{\partial N[(g_1\ x)/y_1, (g_2\ x)/y_2, ...(g_n\ x)/y_n]}{\partial x}|_{t_1} * t
\end{aligned}$$

$$\begin{aligned}
&\tfrac{\partial f\ y}{\partial y}|_{(g\ t_1)} * (\tfrac{\partial g\ z}{\partial z}|_{t_1} * t) \\
&\quad = \tfrac{\partial f\ y}{\partial y}|_{(g_1\ t_1, g_2\ t_1, ..., g_n\ t_1)} * (\tfrac{\partial (g_1\ z, g_2\ z, ..., g_n\ z)}{\partial z}|_{t_1} * t) \\
&\quad = \tfrac{\partial f\ y}{\partial y}|_{(g_1\ t_1, g_2\ t_1, ..., g_n\ t_1)} * (\tfrac{\partial g_1\ z}{\partial z}|_{t_1} * t, \tfrac{\partial g_2\ z}{\partial z}|_{t_1} * t, ..., \tfrac{\partial g_n\ z}{\partial z}|_{t_1} * t) \\
&\quad = \tfrac{\partial (\lambda y_1{:}B.\lambda y_2{:}B.,...\lambda y_n{:}B.N)\ \pi_1(y)\ \pi_2(y)...\ \pi_n(y)}{\partial y}|_{(g_1\ t_1, g_2\ t_1, ..., g_n\ t_1)} * \\
&\qquad\qquad (\tfrac{\partial g_1\ z}{\partial z}|_{t_1} * t, \tfrac{\partial g_2\ z}{\partial z}|_{t_1} * t, ..., \tfrac{\partial g_n\ z}{\partial z}|_{t_1} * t) \\
&\quad = (\tfrac{\partial N[y_1'/y_1, g_2\ t_1/y_2, ..., g_n\ t_1/y_n]}{\partial y_1'}|_{g_1\ t_1}, ..., \tfrac{\partial N[g_1\ t_1/y_1, g_2\ t_1/y_2, ..., y_n'/y_n]}{\partial y_n'}|_{g_n\ t_1}) * \\
&\qquad\qquad (\tfrac{\partial g_1\ z}{\partial z}|_{t_1} * t, \tfrac{\partial g_2\ z}{\partial z}|_{t_1} * t, ..., \tfrac{\partial g_n\ z}{\partial z}|_{t_1} * t) \\
&\quad = (\tfrac{\partial N[y_1'/y_1, g_2\ t_1/y_2, ..., g_n\ t_1/y_n]}{\partial y_1'}|_{g_1\ t_1} * (\tfrac{\partial g_1\ z}{\partial z}|_{t_1} * t)) \oplus ... \oplus \\
&\qquad\qquad (\tfrac{\partial N[g_1\ t_1/y_1, g_2\ t_1/y_2, ..., y_n'/y_n]}{\partial y_n'}|_{g_n\ t_1} * (\tfrac{\partial g_n\ z}{\partial z}|_{t_1} * t))
\end{aligned}$$

Notice that by the base type interpretation, $f(g_1(x), g_2(x), ..., g_n(x)) = f_1'(g_1(x), g_2(x), ..., g_n(x)) * g_1'(x) + f_2'(g_1(x), g_2(x), ..., g_n(x)) * g_2'(x) + ... + f_n'(g_1(x), g_2(x), ..., g_n(x)) * g_n'(x)$ where f_j' means the derivative of f to its j-th parameter, so we get the following and prove the case.

$$\begin{aligned}
&\tfrac{\partial N[(g_1\ x)/y_1, (g_2\ x)/y_2, ...(g_n\ x)/y_n]}{\partial x}|_{t_1} * t \\
&\quad = (\tfrac{\partial N[y_1'/y_1, g_2\ t_1/y_2, ..., g_n\ t_1/y_n]}{\partial y_1'}|_{g_1\ t_1} * (\tfrac{\partial g_1\ z}{\partial z}|_{t_1} * t)) \oplus ... \oplus \\
&\qquad (\tfrac{\partial N[g_1\ t_1/y_1, g_2\ t_1/y_2, ..., y_n'/y_n]}{\partial y_n'}|_{g_n\ t_1} * (\tfrac{\partial g_n\ z}{\partial z}|_{t_1} * t))
\end{aligned}$$

- Case T_2 is base type, T_1 is any type, T is $A \to B$. We prove that for any u of type A, we have $(\tfrac{\partial f(g\ x)}{\partial x}|_{t_1} * t)\ u = (\tfrac{\partial f\ y}{\partial y}|_{(g\ t_1)} * (\tfrac{\partial g\ z}{\partial z}|_{t_1} * t))\ u$.
  First, let $f' = \lambda x : T_1.(f\ x)\ u$, $g' = g$, then by induction we have

  $$\tfrac{\partial f'(g'\ x)}{\partial x}|_{t_1} * t = \tfrac{\partial f'\ y}{\partial y}|_{(g'\ t_1)} * (\tfrac{\partial g'\ z}{\partial z}|_{t_1} * t)$$

  that is, we have

  $$(\tfrac{\partial f\ (g\ x)\ u}{\partial x}|_{t_1} * t) = (\tfrac{\partial f\ y\ u}{\partial y}|_{(g\ t_1)} * (\tfrac{\partial g\ z}{\partial z}|_{t_1} * t))$$

  Then, we prove $(\tfrac{\partial f\ (g\ x)\ u}{\partial x}|_{t_1} * t) = (\tfrac{\partial f\ (g\ x)}{\partial x}|_{t_1} * t)\ u$ by the following calculation.

  $$\begin{aligned}
  (\tfrac{\partial f\ (g\ x)}{\partial x}|_{t_1} * t)\ u &= (\tfrac{\partial \lambda a{:}A.(f\ (g\ x))\ a}{\partial x}|_{t_1} * t)\ u \\
  &= (\lambda a : A.(\tfrac{\partial (f\ (g\ x))\ a}{\partial x}|_{t_1} * t))\ u \\
  &= (\tfrac{\partial f\ (g\ x)\ u}{\partial x}|_{t_1} * t)
  \end{aligned}$$

Next, we prove $(\frac{\partial f\ y}{\partial y}|_{(g\ t_1)} * (\frac{\partial g\ z}{\partial z}|_{t_1} * t))\ u = \frac{\partial f\ y\ u}{\partial y}|_{(g\ t_1)} * (\frac{\partial g\ z}{\partial z}|_{t_1} * t)$. For simplicity, we assume T_1 to be $(B, B, B, ..., B)$ of n-tuples (the technique below can be applied to any T_1 type which makes the term well-typed).
On one hand, by substituting $(g_1\ z, g_2\ z, ..., g_n\ z)$ for $g\ z$, we have

$$\begin{aligned}
&(\frac{\partial f\ y}{\partial y}|_{(g\ t_1)} * (\frac{\partial g\ z}{\partial z}|_{t_1} * t))\ u \\
&\quad = \frac{\partial f\ y}{\partial y}|_{(g_1\ t_1, g_2\ t_1, ..., g_n\ t_1)} * (\frac{\partial (g_1\ z, g_2\ z, ..., g_n\ z)}{\partial z}|_{t_1} * t)\ u \\
&\quad = (\frac{\partial f(y_1, g_2\ t_1, ..., g_n\ t_1)}{\partial y_1'}|_{g_1\ t_1} * (\frac{\partial g_1\ z}{\partial z}|_{t_1} * t) \oplus ... \oplus \\
&\qquad\qquad \frac{\partial f(g_1\ t_1, g_2\ t_1, ..., y_n)}{\partial y_n'}|_{g_n\ t_1} * (\frac{\partial g_n\ z}{\partial z}|_{t_1} * t))\ u
\end{aligned}$$

Since

$$\begin{aligned}
&f(g_1\ t_1, g_2\ t_1, ..., g_{j-1}\ t_1, y_j, g_{j+1}\ t_1, ..., g_n\ t_1) \\
&\quad = \lambda a : A. f\ (g_1\ t_1, g_2\ t_1, ..., g_{j-1}\ t_1, y_j, g_{j+1}\ t_1, ..., g_n\ t_1)\ a
\end{aligned}$$

which will be denoted as $\lambda a : A.t_j^*$, we continue the calculation as follows.

$$\begin{aligned}
&(\frac{\partial f\ y}{\partial y}|_{(g\ t_1)} * (\frac{\partial g\ z}{\partial z}|_{t_1} * t))\ u \\
&\quad = (\frac{\partial \lambda a : A.t_1^*}{\partial y_1}|_{g_1\ t_1} * (\frac{\partial g_1\ z}{\partial z}|_{t_1} * t) \oplus ... \oplus \frac{\partial \lambda a : A.t_n^*}{\partial y_n}|_{g_n\ t_1} * (\frac{\partial g_n\ z}{\partial z}|_{t_1} * t))\ u \\
&\quad = \lambda a : A.(\frac{\partial t_1^*}{\partial y_1}|_{g_1\ t_1} * (\frac{\partial g_1\ z}{\partial z}|_{t_1} * t)) \oplus ... \oplus (\frac{\partial t_n^*}{\partial y_n}|_{g_n\ t_1} * (\frac{\partial g_n\ z}{\partial z}|_{t_1} * t))\ u \\
&\quad = \frac{\partial t_1^*[u/a]}{\partial y_1}|_{g_1\ t_1} * (\frac{\partial g_1\ z}{\partial z}|_{t_1} * t) \oplus ... \oplus \frac{\partial t_n^*[u/a]}{\partial y_n}|_{g_n\ t_1} * (\frac{\partial g_n\ z}{\partial z}|_{t_1} * t)
\end{aligned}$$

On the other hand, we have

$$\begin{aligned}
&(\frac{\partial f\ y\ u}{\partial y}|_{(g\ t_1)} * (\frac{\partial g\ z}{\partial z}|_{t_1} * t)) \\
&\quad = \frac{\partial f\ y\ u}{\partial y}|_{(g_1\ t_1, g_2\ t_1, ..., g_n\ t_1)} * (\frac{\partial (g_1\ z, g_2\ z, ..., g_n\ z)}{\partial z}|_{t_1} * t) \\
&\quad = \frac{\partial f\ (y_1, g_2\ t_1, ..., g_n\ t_1)\ u}{\partial y_1}|_{g_1\ t_1} * (\frac{\partial g_1\ z}{\partial z}|_{t_1} * t) \oplus ... \oplus \\
&\qquad\qquad \frac{\partial f\ (g_1\ t_1, g_2\ t_1, ..., y_n)\ u}{\partial y_n}|_{g_n\ t_1} * (\frac{\partial g_n\ z}{\partial z}|_{t_1} * t) \\
&\quad = \frac{\partial (\lambda a : A.t_1^*)\ u}{\partial y_1}|_{g_1\ t_1} * (\frac{\partial g_1\ z}{\partial z}|_{t_1} * t) \oplus ... \oplus \frac{\partial (\lambda a : A.t_n^*)\ u}{\partial y_n}|_{g_n\ t_1} * (\frac{\partial g_n\ z}{\partial z}|_{t_1} * t) \\
&\quad = (\frac{\partial t_1^*[u/a]}{\partial y_1}|_{g_1\ t_1} * (\frac{\partial g_1\ z}{\partial z}|_{t_1} * t)) \oplus ... \oplus (\frac{\partial t_n^*[u/a]}{\partial y_n}|_{g_n\ t_1} * (\frac{\partial g_n\ z}{\partial z}|_{t_1} * t))
\end{aligned}$$

Therefore, we have proven the case.

- Case T_2 is base type, T_1 is any type, T is $(T_1, T_2, ..., T_n)$. We need to prove that for all j, we have $\pi_j(\frac{\partial f(g\ x)}{\partial x}|_{t_1} * t) = \pi_j(\frac{\partial f\ y}{\partial y}|_{(g\ t_1)} * (\frac{\partial g\ z}{\partial z}|_{t_1} * t))$. We may follow the proof for the case when T has type $A \to B$. Let $f' = \lambda x : T_1.\pi_j(f\ x), g' = g$, by induction, we have

$$\frac{\partial \pi_j(f(g\ x))}{\partial x}|_{t_1} * t = \frac{\partial \pi_j(f\ y)}{\partial y}|_{(g\ t_1)} * (\frac{\partial g\ z}{\partial z}|_{t_1} * t)$$

  The rest of the proof is similar to that for the case when $T = A \to B$.
- Case T_2 is base type, T_1 is any type, T is $T_1 + T_2$. Notice that T_1 has to be base type to be well-typed. But either the case, the proof is similar to the case when $T = A \to B$.
- Case T_2 , T_1 and T are any type. Notice that T_2 does not contain no $\to$ or $+$ to be well-typed (i.e., no derivative over function types). We have proved the case when T_2 is base type, and we assume that T_2 has type $(T_1, T_2, ..., T_n)$. Suppose the normal form of t_1 is $(t_{11}', t_{12}', ..., t_{1n}')$ and the normal form of t is $(t_{21}', t_{22}', ..., t_{2n}')$, Then

$$\begin{aligned}
&\frac{\partial f(g\ x)}{\partial x}|_{t_1} * t \\
&\quad = \frac{\partial f(g\ x)}{\partial x}|_{(t_{11}', t_{12}', ..., t_{1n}')} * (t_{21}', t_{22}', ..., t_{2n}') \\
&\quad = (\frac{\partial f(g\ (x_1, t_{12}', ..., t_{1n}'))}{\partial x_1}|_{t_{11}'}, ..., \frac{\partial f(g\ (t_{11}', t_{12}', ..., x_n))}{\partial x_n}|_{t_{1n}'}) * (t_{21}', ..., t_{2n}') \\
&\quad = (\frac{\partial f(g\ (x_1, t_{12}', ..., t_{1n}'))}{\partial x_1}|_{t_{11}'} * t_{21}') \oplus ... \oplus (\frac{\partial f(g\ (t_{11}', t_{12}', ..., x_n))}{\partial x_n}|_{t_{1n}'} * t_{2n}')
\end{aligned}$$

On the other hand, we can use Lemma 9 (i.e., $t_1 * (t_2 \oplus t_3) = (t_1 * t_2) \oplus (t_1 * t_3)$) to do the following calculation.

$$\begin{aligned}
&\frac{\partial f\ y}{\partial y}|_{(g\ t_1)} * (\frac{\partial g\ z}{\partial z}|_{t_1} * t) \\
&\quad = \frac{\partial f\ y}{\partial y}|_{(g\ t_1)} * ((\frac{\partial g\ (x_1, t'_{12}, \ldots, t'_{1n})}{\partial x_1}|_{t'_{11}} * t'_{21}) \oplus \ldots \oplus (\frac{\partial g\ (t'_{11}, t'_{12}, \ldots, x_n)}{\partial x_n}|_{t'_{1n}} * t'_{2n})) \\
&\quad = \frac{\partial f\ y}{\partial y}|_{(g\ t_1)} * (\frac{\partial g\ (x_1, t'_{12}, \ldots, t'_{1n})}{\partial x_1}|_{t'_{11}} * t'_{21}) \oplus \ldots \oplus \\
&\qquad\quad \frac{\partial f\ y}{\partial y}|_{(g\ t_1)} * (\frac{\partial g\ (t'_{11}, t'_{12}, \ldots, x_n)}{\partial x_n}|_{t'_{1n}} * t'_{2n})
\end{aligned}$$

Now by induction using $f' = f, g' = \lambda x : T_j.g\ (t'_{11}, t'_{12}, ..., t'_{1(j-1)}, x, t'_{1(j+1)}, ..., t'_{1n})$, we have

$$\begin{aligned}
&\frac{\partial f(g\ (t'_{11}, t'_{12}, \ldots, t'_{1(j-1)}, x_j, t'_{1(j+1)} \cdots, t'_{1n})}{\partial x_j}|_{t'_{1j}} * t'_{2j} \\
&\quad = \frac{\partial f\ y}{\partial y}|_{(g'\ t'_{1j})} * (\frac{\partial g\ (t'_{11}, t'_{12}, \ldots, t'_{1(j-1)}, x_j, t'_{1(j+1)} \cdots, t'_{1n})}{\partial x_j}|_{t'_{1j}} * t'_{2j}) \\
&\quad = \frac{\partial f\ y}{\partial y}|_{(g\ t_1)} * (\frac{\partial g\ (t'_{11}, t'_{12}, \ldots, t'_{1(j-1)}, x_j, t'_{1(j+1)} \cdots, t'_{1n})}{\partial x_j}|_{t'_{1j}} * t'_{2j})
\end{aligned}$$

Therefore by Lemma 7, we have proven the case.
Thus we have proven the theorem. ◀

Application: Automatic Differentiation

The Chain Rule provides another way to compute the derivatives. There are many applications of the chain rule, and here we give an example of how to associate it with the automatic differentiation [10].

▶ **Example 14** (AD). This is an example from [10]. Let *sqr* and *magSqr* be defined as follows.

$$\begin{array}{lcl}
sqr & :: & \mathbb{R} \to \mathbb{R} \\
sqr\ a & = & a * a \\
magSqr & :: & (\mathbb{R}, \mathbb{R}) \to \mathbb{R} \\
magSqr\ (a, b) & = & sqr\ a \oplus sqr\ b
\end{array}$$

First of all, let t_1 and t_2 two pairs, then it is easy to prove that $\frac{\partial (t_1 \oplus t_2)}{\partial x}|_{t_3} = \frac{\partial t_1}{\partial x}|_{t_3} \oplus \frac{\partial t_2}{\partial x}|_{t_3}$. Next, we can perform automatic differentiation on *magSqr* by the following calculation.

$$\begin{aligned}
&\frac{\partial (magSqr\ x)}{\partial x}|_{(a,b)} * t \\
&\quad = \frac{\partial (sqr(\pi_1 x) \oplus sqr(\pi_2 x))}{\partial x}|_{(a,b)} * t \\
&\quad = \frac{\partial (sqr\ y)}{\partial y}|_{\pi_1(a,b)} * (\frac{\partial (\pi_1 x)}{\partial x}|_{(a,b)} * t) \oplus \frac{\partial (sqr\ y)}{\partial y}|_{\pi_2(a,b)} * (\frac{\partial (\pi_2 x)}{\partial x}|_{(a,b)} * t) \\
&\quad = 2 * a * ((1, 0) * t) \oplus 2 * b * ((0, 1) * t)
\end{aligned}$$

Now, because the theorem applies for any t of pair type, we use $(1, 0)$ and $(0, 1)$ to substitute for t respectively, and we will get $\frac{\partial (magSqr\ x)}{\partial x}|_{(a,b)} = (2 * a, 2 * b)$, which means its derivative to a is $2 * a$ and its derivative to b is $2 * b$.

5 Taylor's Theorem

In this section, we discuss Taylor's Theorem, which is useful to give an approximation of a k-order differentiable function around a given point by a polynomial of degree k. In programming, it is important and has many applications in approximation and incremental computation.

First of all, we introduce some high-order notations.

$$\begin{array}{lcllcl} \frac{\partial^0 t_1}{\partial x^0}|_{t_2} & = & t_1 & \frac{\partial^n t_1}{\partial x^n}|_{t_2} & = & \frac{\partial \frac{\partial^{n-1} t_1}{\partial x^{n-1}}|_x}{\partial x}|_{t_2} \\ t * t_1^0 & = & t & t * t_1^n & = & (t * t_1) * t_1^{n-1} \\ f^0 & = & f & f^n & = & (f')^{n-1} \\ (\lambda x : T.\, t)' & = & \lambda x : T.\, \frac{\partial t}{\partial x}|_x & & & \end{array}$$

Now our Taylor's Theorem can be expressed as follows.

▶ **Theorem 15** (Taylor's Theorem). If both $f\ t$ and $\sum_{k=0}^{\infty} \frac{1}{k!}(f^{(k)}\ t_0) * (t \ominus t_0)^k$ are weak-normalizable, then

$$f\ t = \sum_{k=0}^{\infty} \frac{1}{k!}(f^{(k)}\ t_0) * (t \ominus t_0)^k.$$

Proof. Like in the proof of Theorem 11, for simplicity, we assume that f, g, t and t_1 are closed. Furthermore, we assume that t and t_1 are in normal form. We prove it by induction on the type of $f : T \to T'$.

- Case T' is a base type. T must contain no $\to$ by our typing, so for simplicity, we suppose T to be $(B, B, ..., B)$. Using the same technique in Theorem 13, we assume f to be

$$f = \lambda x : T.\, (\lambda x_1 : B.\, \lambda x_2 : B., ... \lambda x_n : B.\, N)\ \pi_1(x)\ \pi_2(x) ...\ \pi_n(x)$$

(denoted by $f = \lambda x : T.\, t_2$ later), t to be $(t_{11}, t_{12}, ..., t_{1n})$, and t_0 to be $(t_{21}, t_{22}, ..., t_{2n})$, where each t_{ij} is a normal form of base type. Then we have

$$\begin{aligned} &(f^{(n)}\ t_0) * (t \ominus t_0)^n \\ &\quad = \frac{\partial^n t_2}{\partial x^n}|_{t_0} * (t \ominus t_0)^n \\ &\quad = (\frac{\partial \frac{\partial^{n-1} t_2}{\partial x^{n-1}}|_{(x_1, t_{22}, ..., t_{2n})}}{\partial x_1}|_{t_{21}}, ..., \frac{\partial \frac{\partial^{n-1} t_2}{\partial x^{n-1}}|_{(t_{21}, t_{22}, ..., x_n)}}{\partial x_n}|_{t_{2n}}) * (t \ominus t_0)^n \\ &\quad = (\frac{\partial \frac{\partial^{n-1} t_2}{\partial x^{n-1}}|_{(x_1, t_{22}, ..., t_{2n})}}{\partial x_1}|_{t_{21}} * (t_{11} \ominus t_{21}) \oplus ... \oplus \frac{\partial \frac{\partial^{n-1} t_2}{\partial x^{n-1}}|_{(t_{21}, t_{22}, ..., x_n)}}{\partial x_n}|_{t_{2n}} \\ &\qquad\qquad * (t_{1n} \ominus t_{2n})) * (t \ominus t_0)^{n-1} \\ &\quad = ((\frac{\partial (\frac{\partial \frac{\partial^{n-2} t_2}{\partial x^{n-2}}|_{(x_1, t_{22}, ..., t_{2n})}}{\partial x_1}|_{x_1}, ..., \frac{\partial \frac{\partial^{n-2} t_2}{\partial x^{n-2}}|_{(x_1, t_{22}, ..., x_n)}}{\partial x_n}|_{t_{2n}})}{\partial x_1}|_{t_{21}}) * (t_{11} \ominus t_{21}) \oplus ... \oplus \\ &\qquad\qquad (\frac{\partial (\frac{\partial \frac{\partial^{n-2} t_2}{\partial x^{n-2}}|_{(x_1, t_{22}, ..., x_n)}}{\partial x_1}|_{t_{21}}, ..., \frac{\partial \frac{\partial^{n-2} t_2}{\partial x^{n-2}}|_{(t_{21}, t_{22}, ..., x_n)}}{\partial x_n}|_{x_n})}{\partial x_n}|_{t_{2n}}) \\ &\qquad\qquad * (t_{1n} \ominus t_{2n})) * (t \ominus t_0)^{n-1} \\ &\quad = ((\frac{\partial \frac{\partial \frac{\partial^{n-2} t_2}{\partial x^{n-2}}|_{(x_1, t_{22}, ..., t_{2n})}}{\partial x_1}|_{x_1}}{\partial x_1}|_{t_{21}}) * (t_{11} \ominus t_{21})^2 \oplus ... \oplus \\ &\qquad\qquad ((\frac{\partial \frac{\partial \frac{\partial^{n-2} t_2}{\partial x^{n-2}}|_{(x_1, t_{22}, ..., x_n)}}{\partial x_1}|_{t_{21}}}{\partial x_n}|_{t_{2n}}) * (t_{1n} \ominus t_{2n})) * (t_{11} \ominus t_{21}), \end{aligned}$$

$$
\begin{aligned}
&((\frac{\partial \frac{\partial^{n-2} t_2}{\partial x^{n-2}}|_{(x_1,x_2,\ldots,t_{2n})}}{\partial x_2}|_{t_{22}}}{\partial x_1}|_{t_{21}}) * (t_{11} \ominus t_{21})) * (t_{12} \ominus t_{22}) \oplus ... \oplus \\
&\qquad ((\frac{\partial \frac{\partial^{n-2} t_2}{\partial x^{n-2}}|_{(t_{11},x_2,\ldots,x_n)}}{\partial x_2}|_{t_{22}}}{\partial x_n}|_{t_{2n}}) * (t_{1n} \ominus t_{2n})) * (t_{12} \ominus t_{22}), \\
&... \\
&((\frac{\partial \frac{\partial^{n-2} t_2}{\partial x^{n-2}}|_{(x_1,t_{22},\ldots,x_n)}}{\partial x_n}|_{t_{2n}}}{\partial x_1}|_{t_{21}}) * (t_{11} \ominus t_{21})) * (t_{1n} \ominus t_{2n}) \oplus ... \oplus \\
&\qquad (\frac{\partial \frac{\partial^{n-2} t_2}{\partial x^{n-2}}|_{(t_{11},t_{22},\ldots,x_n)}}{\partial x_n}|_{x_n}}{\partial x_n}|_{t_{2n}}) * (t_{1n} \ominus t_{2n})^2) * (t \ominus t_0)^{n-2} \\
&= ...
\end{aligned}
$$

As seen in the above, every time we decompose a $\frac{\partial}{\partial x_i}|_{(...)}$, apply Rule EAppDer1, and then make reduction with Rule EAppMul3 to lower down the exponent of $(t \ominus t_0)^n$. Finally, we will decompose the last derivative and get the term t_2 in the form of $t_2[t'_{21}/x_1, t'_{22}/x_2, ..., t'_{2n}/x_n]$ where $\forall j \in [1, n], t'_{2j}$ is either t_{2j} or x_j.
Note that on base type we assume that we have Taylor's Theorem:

$$f(x_0 + h) = f(x_0) + \sum_{k=1}^{\infty} \frac{1}{k!} (\sum_{i=1}^{n} h_i \frac{\partial}{\partial x_i})^k f(x_0)$$

where x_0 and h is an n-dimensional vector, and x_j, h_j is its projection to its j-th dimension.
So we have $(f^{(k)}\ t_0) * (t \ominus t_0)^k$ corresponds to the k-th addend $\frac{1}{k!}(\sum_{i=1}^{n} h_i \frac{\partial}{\partial x_i})^k f(x_0)$.

- Case: T' is function type $A \to B$. Similar to the proof in Theorem 13, for all u of type A, we define $f^* = \lambda x : T.\, f\ x\ u$, and by using the inductive result on type B, we can prove the case simiarly as that in Theorem 13.
- Case: T' is a tuple type $(T_1, T_2, T_3, ...)$. Just define $f^* = \lambda x : T.\, \pi_j(f\ x)$ to use inductive result. The rest is simple.
- Case: T' is a tuple type $T_1 + T_2$. This case is impossible because the righthand is not well-typed.

Thus we have proven the theorem. ◀

Application: Polynomial Approximation

Taylor's Theorem has many applications. Here we give an example of using Taylor's Theorem for approximation. Suppose there is a point $(1, 0)$ in the polar coordinate system, and we want to know where the point will be if we slightly change the radius r and the angle θ. Since it is extremely costive to compute functions such as sin() and cos(), Taylor's Theorem enables us to make a fast polynomial approximation.

▶ **Example 16.** Let function *polar2catesian* be defined by

$$
\begin{array}{lcl}
\mathit{polar2cartesian} & :: & (\mathbb{R}, \mathbb{R}) \to (\mathbb{R}, \mathbb{R}) \\
\mathit{polar2cartesian}(r, \theta) & = & (r * \cos(\theta), r * \sin(\theta))
\end{array}
$$

We show how to expand $\mathit{polar2cartesian}(r, \theta)$ at $(1, 0)$ up to 2nd-order derivative. Since

$$\begin{aligned}\frac{\partial(\mathit{polar2cartesian}(x))}{\partial x}|_{(1,0)} &= \frac{\partial(\pi_1 x * \mathit{cos}(\pi_2 x), \pi_1 x * \mathit{sin}(\pi_2 x)}{\partial x}|_{(1,0)} \\ &= (\frac{\partial(x_1 * \cos(0), x_1 * \sin(0))}{\partial x_1}|_1, \frac{\partial(1 * \cos(x_2), 1 * \sin(x_2))}{\partial x_2}|_0) \\ &= ((1, 0), (0, 1))\end{aligned}$$

we have

$$\frac{\partial(\mathit{polar2cartesian}(x))}{\partial x}|_{(1,0)} * (\Delta r, \Delta\theta) = (\Delta r, \Delta\theta).$$

Again, we have

$$\begin{aligned}\frac{1}{2}\frac{\partial^2(\mathit{polar2cartesian}(x))}{\partial x^2}|_{(1,0)} * (\Delta r, \Delta\theta)^2 &= (((0, 0), (0, 1)), ((0, 1), (-1, 0))) * (\Delta r, \Delta\theta)^2 \\ &= (-\frac{1}{2}\Delta\theta^2, \Delta r * \Delta\theta).\end{aligned}$$

Combining the above, we can use $(1 \oplus \Delta r \ominus \frac{1}{2}\Delta\theta^2, \Delta\theta \oplus \Delta r * \Delta\theta)$ to make an approximation to $\mathit{polar2cartesian}(1 + \Delta r, \Delta\theta)$.

6 Related Work

Differential Calculus and The Change Theory. The differential lambda-calculus [9, 8] has been studied for computing derivatives of arbitrary higher-order programs. In the differential lambda-calculus, derivatives are guaranteed to be linear in its argument, where the incremental lambda-calculus does not have this restriction. Instead, it requires that the function should be differentiable. The big difference between our calculus and differential lambda calculus is that we perform computation on terms instead of analysis on terms.

The idea of performing incremental computation using derivatives has been studied by Cai et al. [7], who give an account using change structures. They use this to provide a framework for incrementally evaluating lambda calculus programs. It is shown that the work can be enriched with recursion and fix-point computation [4]. The main difference between our work and change theory is that we describe changes as mathematical derivatives while the change theory describe changes as (discrete) deltas.

Incremental/Self-Adaptive Computation. Paige and Koenig [19] present derivatives for a first-order language with a fixed set of primitives for incremental computation. Blakeley et al. [16] apply these ideas to a class of relational queries. Koch [14] guarantees asymptotic speedups with a compositional query transformation and delivers huge speedups in realistic benchmarks, though still for a first-order database language. We have proved Taylor's theorem in our framework, which provides us with another way to perform finite difference on the computation.

Self-adjusting computation [2] or adaptive function programming [3] provides a dynamic approach to incrementalization. In this approach, programs execute on the original input in an enhanced runtime environment that tracks the dependencies between values in a dynamic dependence graph; intermediate results are memoized. Later, changes to the input propagate through dependency graphs from changed inputs to results, updating both intermediate and final results; this processing is often more efficient than recomputation. Mathematically, self-adjusting computations corresponds to differential equations (The derivative of a function can be represented by the computational result of function), which may be a future work of our calculus.

Automatic Differentiation. Automatic differentiation [12] is a technique that allows for efficiently computing the derivative of arbitrary programs, and can be applied to probabilistic modeling [15] and machine learning [6]. This technique has been successfully applied to some higher-order languages [21, 10]. As pointed out in [4], while some approaches have been suggested [18, 13], a general theoretical framework for this technique is still a matter of open research. We prove the chain rule inside our framework, which lays a foundation for our calculus to perform automatic differentiation. And with more theorems in our calculus, we expect more profound applications in differential calculus.

7 Conclusion

In this paper, we propose an analytical differential calculus which is equipped with integration. This calculus, as far as we are aware, is the first one that has well-defined integration, which has not appeared in both differential lambda calculus and the change theory. Our calculus enjoys many nice properties such as soundness and strong normalizing, and has three important theorems, which have profound applications in computer science. Also, our calculus is highly extendable, it would be easy for users to add new features and prove more theorems inside our calculus.

References

1 Martín Abadi and Gordon D. Plotkin. A simple differentiable programming language. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–28, January 2020. doi:10.1145/3371106.

2 Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative self-adjusting computation. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 309–322. ACM, 2008.

3 Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In John Launchbury and John C. Mitchell, editors, *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 247–259. ACM, 2002.

4 Mario Alvarez-Picallo, Alex Eyers-Taylor, Michael Peyton Jones, and C.-H. Luke Ong. Fixing incremental computation - derivatives of fixpoints, and the recursive semantics of datalog. In Luís Caires, editor, *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 525–552. Springer, 2019.

5 Mario Alvarez-Picallo and C. H. Luke Ong. The difference lambda-calculus: A language for difference categories, 2020. arXiv:2011.14476.

6 Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *J. Mach. Learn. Res.*, 18:153:1–153:43, 2017.

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 Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 145–155. ACM, 2014.

8 Thomas Ehrhard. An introduction to differential linear logic: proof-nets, models and antiderivatives. *Math. Struct. Comput. Sci.*, 28(7):995–1060, 2018.

9 Thomas Ehrhard and Laurent Regnier. The differential lambda-calculus. *Theor. Comput. Sci.*, 309(1-3):1–41, 2003.

10 Conal Elliott. The simple essence of automatic differentiation. *Proc. ACM Program. Lang.*, 2(ICFP):70:1–70:29, 2018.

11 Paolo G. Giarrusso, Yann Régis-Gianas, and Philipp Schuster. Incremental \lambda -calculus in cache-transfer style - static memoization by program transformation. In Luís Caires, editor, *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 553–580. Springer, 2019.

12 Andreas Griewank and Andrea Walther. *Evaluating derivatives - principles and techniques of algorithmic differentiation, Second Edition.* SIAM, 2008.

13 Robert Kelly, Barak A. Pearlmutter, and Jeffrey Mark Siskind. Evolving the incremental λ calculus into a model of forward automatic differentiation (AD). *CoRR*, abs/1611.03429, 2016. `arXiv:1611.03429`.

14 Christoph Koch. Incremental query evaluation in a ring of databases. In Jan Paredaens and Dirk Van Gucht, editors, *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2010, June 6-11, 2010, Indianapolis, Indiana, USA*, pages 87–98. ACM, 2010.

15 Alp Kucukelbir, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David M. Blei. Automatic differentiation variational inference. *J. Mach. Learn. Res.*, 18:14:1–14:45, 2017.

16 Per-Åke Larson and Jingren Zhou. Efficient maintenance of materialized outer-join views. In Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis, editors, *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 56–65. IEEE Computer Society, 2007.

17 Yanhong A. Liu. Efficiency by incrementalization: An introduction. *High. Order Symb. Comput.*, 13(4):289–313, 2000. `doi:10.1023/A:1026547031739`.

18 Oleksandr Manzyuk. A simply typed λ-calculus of forward automatic differentiation. *Electronic Notes in Theoretical Computer Science*, 286:257–272, 2012. Proceedings of the 28th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVIII).

19 Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3):402–454, 1982.

20 Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 2002.

21 Jeffrey Mark Siskind and Barak A. Pearlmutter. Nesting forward-mode AD in a functional framework. *High. Order Symb. Comput.*, 21(4):361–376, 2008.