Deterministic differential calculi

Marie Kerjean
IRIF, Université Paris Diderot, Paris
marie.kerjean@pps.univ-paris-diderot.fr

Abstract

Differential lambda-calculus is an extension of lambda-calculus that includes a notion of differentiation on terms. Differentiation introduces formal sums of terms, inducing some non-determinism. In order to reintroduce determinism, we consider two subcalculi of the differential lambda-calculus, corresponding respectively to a call-by-name and a call-by-value interpretation of differentiation. We present a differential version of the Krivine’s abstract machine which computes the differential call-by-name lambda-calculus, and a differential version of the CEK machine which computes the differential call-by-value lambda-calculus. We synthesize these calculi into a differential enriched call-by-push-value.

1 Introduction

Functional programming is the art of handling programs seen as functions. The very core of functional programming languages is expressed in $\lambda$-calculus [3]. Thus, programs (terms) of $\lambda$-calculus are constructed with the following syntax: they either are variables $x, y, \ldots$, functions (abstractions) $\lambda z.t$ mapping a value for $z$ to $t$, or the application of two terms $tu$. Moreover, in typed $\lambda$-calculus different kinds of terms are distinguished, so as to ensure their well-behaviour, as the termination of reduction. Meanwhile, in denotational semantics one tightens the function-as-program relation by interpreting types as spaces in a given mathematical structure, and terms as morphisms between those spaces. In fact, these models interpret also the logical system typing $\lambda$-calculus: intuitionistic logic. In the 80’s, Girard refined intuitionistic logic into linear logic [14]. As a consequence, many models of typed $\lambda$-calculus arise from models of linear logic.

In several cases, there in these models is a notion of differentiation on functions [15, 8, 4]. In a function-as-program spirit, differential $\lambda$-calculus [9] puts together the notion of differentiation and the one of functional program. Thus, differential $\lambda$-calculus formalizes a way to linearize a lambda-term by a syntactic differentiation operator.

This differentiation operator goes along with the introduction of sums of terms. We want to get a better grasp of these sums, as it will open the way to constructing the reverse operation: a syntactic integration. Moreover, it is still difficult to see what are the effects of differentiation on real-life programs. We hope that, by precising the subterms of these sums, it will emerge new intuitions on that matter.

Remember that in differential geometry, the differentiation of a function is understood as its local linear approximation. We thus argue that the differentiation of a lambda-term should be the linear approximation of this term with respect to its local behaviour. That is, the differentiation of a term should take into account the reduction path followed by a term at every stage of its evaluation. We thus study differentiation in languages equipped with a reduction strategy.

**Linearity in the syntax** In algebraic calculus, the evaluation map $f, x \mapsto f(x)$ is linear in $f$ but not in $x$. Indeed $(f + g)(x) = f(x) + g(x)$ while $f(x + y) = f(x) + f(y)$ if and only if $f$ is a linear function.

The interpretation of linearity in the $\lambda$-calculus follows this intuition: in the application $tu$, $t$ is linear but $u$ is not. That is, a variable $x$ is linear in a term $t$ if and only if it is in head position in $t$: $t = xt'$ for some term $t'$ not containing $x$. This notion of linearity can be grasped in term of resources. During the computation of $tu$, the value of $t$ will be looked for only once, while the value of $u$ could be used more than once. On one hand,

$$(\lambda x.xy)z$$
is linear both in $(\lambda x.xy)$ and in $z$: it will reduce to $zy$ which is linear in $z$. On the other hand, the term

$$(\lambda x.fx)z$$

is linear in $(\lambda x.fx)$ but not in $z$, because $z$ will be used more than once during the computation of $fzz$.

**Differential $\lambda$-calculus** The differential $\lambda$-calculus was introduced by Ehrhard and Regnier [9] as a syntactic account for the mathematical theory of differential calculus. To the terms of $\lambda$-calculus is added a differential application $Ds \cdot u$ which represents the term $s$ linearly applied to $u$. That is, $u$ should be used only once in $Ds \cdot u$. This is why the authors introduce a new reduction rule for this differential application:

$$(D\lambda z.t) \cdot u \rightarrow_{\beta_D} \lambda z. (\partial z.t)u.$$  

This rule is alike the $\beta$ rule of $\lambda$-calculus:

$$(\lambda z.t).u \rightarrow_{\beta} t[u/z].$$

The newly introduced $(\partial z.t)u$ is the linear substitution of $z$ by $u$ in $t$. It is an inductively defined substitution where one chooses to replace a unique linear occurrences of $z$ in $t$. As this supposes that not all occurrences of $z$ are replaced at the same time, $z$ is still free in $(\partial z.t)u$, and thus $(D\lambda z.t).u$ reduces to a function of $z$.

The emblematic rule for linear substitution is the one defining it on an application. It introduces a choice through a sum:

$$(\partial z.su)t = ((\partial z.s)t)u + (Ds \cdot ((\partial z.u)t))u$$  \hspace{1cm} (1)

If $z$ is linear in $s$, then so it is in $sv$. To substitute linearly $z$ by $u$ in $sv$, we can then substitute it linearly in $s$ and then apply the result to $v$. But we can also look for a linear occurrence of $z$ in $v$. In that case, for $v$ to remain linear in $(\partial z.sv)u$, we should linearize $s$ before applying it to $(\partial z.v)u$. Then $s$ will be fed by a linear copy of $(\partial z.v)u$, and then it will be fed by $u$ as usual. This last case can be seen as a computational interpretation for the chain rule in differential calculus.

**Deterministic differentiation** We argue that the sum in the previous equality really is a sum over the evaluation paths for $sv$. The idea is that in a deterministic version of $(\partial z.t)u$ only the first occurrence of $z$ should be replaced. It means that the linear substitution process will become a dynamic process, following the evaluation of the term. The variable $z$ will be linearly substituted the first time it is encountered during the reduction process. With this point of view, one can see the first summand of formula (1) as a call-by-name case, while the second summand would correspond to a call-by-value strategy.

If a (weak) call-by-name strategy is followed during the whole reduction process, this amounts to substituting only the weak head occurrence of $z$ in $t$. In that case, the evaluation of $(\partial z.t)u$ would be done according to the following rules:

$$\frac{\lambda x.((\partial s.x)u \rightarrow s')}{(\partial s.t.x)u \rightarrow s't}$$

$$\frac{\lambda x.((\partial t.x)u \rightarrow t')}{(\partial z.t)u \rightarrow (\partial z.t')u}$$

This differential call-by-name $\lambda$-calculus can be implemented in a variant of the Krivine’s Abstract Machine (KAM) [16]. It is enough to add a temporary and prioritary global environment $T$ to the KAM. When considering a variable $z$, the machine will first look into $T$ to see if $z$ is bound to a term $t$. If it is the case, it will replace $x$ by $t$ and free $x$ from the domain of $T$. When considering a linear substitution $(\partial z.t)u$, the machine will bound $z$ to $u$ in $T$ and then compute $t$. As the linear substitution of differential $\lambda$-calculus allows reduction under the abstraction, we study a strong version of the deterministic call-by-name $\lambda$-calculus which compute a subterm of the differential $\lambda$-calculus.

The same principles can be applied to a call-by-value reduction strategy: when computing $(\partial z.t)u$ only the first encountered occurrence of $z$ should be replaced by $u$. The key rules of this (weak) differential call-by-value $\lambda$-calculus are the following:
(\partial z.zt)u \rightarrow ut

s \rightarrow s'

(\partial z.sw)u \rightarrow (\partial s'w.t')u

(\partial z.w)u \rightarrow w'

(\partial x.t)w' \rightarrow t'

(\partial z.(\lambda x.t)w)u \rightarrow (\lambda x.t')w

The third rule is the implementation of the chain rule in differential calculus. It corresponds to the second summand in the formula \[.] This calculus can be implemented in a variant of the CEK machine \([15]\). As in the call-by-name case, this is done by adding a temporary, prioritary and global environment to the machine. The handling of \(T\) is however more subtle, due to the last rule above.

Call-by-push value We want to understand differentiation of terms which are not assigned a specific reduction strategy beforehand. This begins by understanding the differential version of a formal programming language in which both differential call-by-name and differential call-by-value \(\lambda\)-calculi embed, such as call-by-push-value \([17]\).

Call-by-push-value is a formal programming language in which two classes of terms cohabit: computations, which are dynamic entities, and values which are static entities. Computations are created from usual terms with a return construction. Indeed, return(\(t\)) is the computation whose dynamic is to wait to be called and then to give \(t\) as an argument. The inverse construction is the thunk operator on computations, whose purpose is to delay the evaluation of the computation \(t\).

The enriched effect calculus \([7]\), denoted as EEC, is a computational language which can be seen as an enrichment with linear types of a call-by-push-value like language. Egger, Mogelberg and Simpson expose a natural notion of linearity for computation: a subcomputation is linear if it is the first one to be performed. This notion of linearity differs from the linearity of resources we just studied. This distinction corresponds to the difference between computations and values in call-by-push-value. A computation is linear in \(t\) if it is performed only once, while a value is linear if it is looked for only once. The study of differential enriched call-by-push-value is the occasion of formalizing this idea.

We consider an enriched call-by-push-value inspired by EEC, that is call-by-push-value with only one additional linear type \(!A\otimes A\). We construct a term return \(v \otimes u\) typed by \(!A\otimes A\) representing a non-linear argument \(v\) of type \(A\) and a linear argument \(u\) of type \(A\). The computation

\[
\text{let } x = \text{return } v \otimes u \text{ in } t
\]

behaves as the differential lambda-term

\[D(\lambda x.t)uv\]

which reduces into

\[(\lambda x.(\partial x.t)u)v.\]

We also study a differentiation operator \(Dt\). It allows to reason about the general notion of differentiation on a term which may not be an abstraction. Following the EEC intuition, we say that one can differentiate a computation by replacing its first subcomputation.

Contributions To summarize, our contribution are:

- We formalize in various context the notion of a deterministic linear substitution.
- We construct a differential (weak) call-by-name \(\lambda\)-calculus, and a differential version of the KAM. We prove the correctness of the KAM with respect to this calculus.
- We detail a strong version of the differential call-by-name \(\lambda\)-calculus, and prove that its linear substitution computes one of the subterms of the differential \(\lambda\)-calculus linear substitution.
- We construct a differential (weak) call-by-value \(\lambda\)-calculus, and a differential version of the CEK machine without control operators. We prove the partial correctness of the differential CEK machine with respect to this calculus.
• We build a strong version of the differential call-by–value λ-calculus, and prove that its linear substitution computes one of the subterms of the differential λ-calculus linear substitution.

• We construct a differential version of call-by-push-value. We implement in this calculus the intuitions of EEC for linearity of computations. We prove that differential call-by-name or call-by–value calculi embed in this calculus.

Related works  This work follows the study of differential λ-calculus by Ehrhard and Regnier [9]. It differs from most of the research on the subject as it is rid of sums and non-determinism. However, the call-by-name and call-by–value duality already triggered studies on the subject. Resource calculus [5, 12] is a calculus with a notion of Linear application. It bears many similarities with the differential λ-calculus and Boudol, Curien and Lavatelli [6] developed for it a call-by-name version very similar to our call-by-name differential λ-calculus, in the sense that linear application will replace the occurrence in head position of a variable. In an article by Ehrhard [11], one can find a notion of call-by-value resource calculus which relates to our differential call-by-value λ-calculus in the extent that variables are only linearly substituted by values.

Moreover, some recent works on calculi inspired by linear logic make use of the distinction between call-by-name and call-by-value [2, 10]. They do not include differentiation in the study though.

Contents of the paper  We recall the syntax and operational semantics of the differential λ-calculus in section 2. Then in section 3 we expose a differential (weak) call-by-name calculus, a strong version of this calculi , and a differential version of the Krivine’s abstract machine. The content of section 4 is symmetric as we expose a differential (weak) call-by–value calculus, a strong version of this calculi, and a differential version of the CEK machine without control operators. Finally, we construct a differential version of call-by-push-value in section 5. We discuss the perspectives of our work in section 6.

2 The differential λ-calculus

We recall the syntax and operational semantics of the differential λ-calculus. As already explained, the linear substitution generates a choice, which is written as a sum. Thus differential λ-calculus needs to deal with sums of terms. We write simple terms as s, t, u, v, w while sums of terms are denoted with S, T, U, V. The set of simple terms is denoted Λs and the set of sums of terms is denoted Λd. They are constructed according to the following syntax:

\[ \Lambda^s : s, t, u, v ::= x \mid \lambda x. s \mid s T \mid D s \cdot t, \]

\[ \Lambda^d : S, T, U, V ::= 0 \mid s \mid s + T. \]

We write \( \lambda x. \sum s_i \) for \( \sum \lambda x. s_i \), \( (\sum s_i)T \) for \( \sum s_i T \), and \( D(\sum s_i) \cdot (\sum t_j) \) for \( \sum s_i t_j \). Sums are of course commutative, and as 0 is the neutral element of the sum, we write S for \( S + 0 \).

Notation. We write \( (\partial x.t)u \) for the linear substitution of \( x \) by \( u \) in \( t \). Beware that it differs from Ehrhard and Regnier’s notation \( \frac{\partial t}{\partial x}. u \).

The reduction process in differential λ-calculus is the smallest reduction relation following the two rules:

\[ (\lambda x.s)T \rightarrow^{\beta} s[T/x] \]

\[ D(\lambda x.s) \cdot t \rightarrow^{\beta_D} \lambda x.(\partial x.s)t \]

which is closed by the usual contextual rules:
A differential call-by-name λ-calculus, making a direct use of linear substitution in differential lambda-calculi, each one in a strong and in a weak version. For sake of clarity, we change slightly in the following sections, we will construct two deterministic differential λ-calculus are:

\[(\partial x.y)T = \begin{cases} T & \text{if } x = y \\ 0 & \text{otherwise} \end{cases} \]

\[(\partial x.\lambda y.s)T = \lambda y.(\partial x.s)T \]

\[(\partial x.0)T = 0 \]

\[(\partial x.s + U)T = (\partial x.s)T + (\partial x.U)T \]

Figure 1: The linear substitution

- If \( s \to S' \) then \( st \to S'T \), \( \lambda x.s \to \lambda x.S' \) and \( Ds \cdot t \to DS' \cdot t \).
- If \( T \to T' \) then \( sT \to sT' \).

We consider moreover the simple terms of differential λ-calculus up to \( \eta \)-reduction: in the term \( \lambda x.s \) \( x \) is supposed to be free in \( s \). It means that when we consider \( (\partial x.s)t \), \( x \) is always supposed to be free in \( s \).

The linear substitution We recall the rules of linear substitution in figure 2. The central and most intricate of them is the one defining linear substitution on an application. Intuitions were given in the introduction, see formula 1. In rule 2, we observe the simplest difference between substitution and linear substitution: while \( y[t/x] = y \), here \( ((\partial x.y))t = 0 \). In rule 6, we observe the version of rule 5 with a linear application. The second summand is thus much simpler, as the function \( Ds \) is already linear it does not need to be linearized as in rule 3.

The differential λ-calculus is typed with the usual rules of the simply typed λ-calculus, to which is added the following:

\[
\frac{\Gamma \vdash u : A \quad \Gamma \vdash s : A \to B}{\Gamma \vdash s \cdot u : A \to B}
\]

3 A differential call-by-name λ-calculus

Linear substitution as a term In the following sections, we will construct two deterministic differential lambda-calculi, each one in a strong and in a weak version. For sake of clarity, we change slightly the language, making a direct use of linear substitution \( (\partial x.) \) without the differntiation operator \( D \). The linear substitution \( (\partial x.t) \) becomes a term of the calculus, and can be understood as a "linear abstraction".

We choose to use the linear substitution as a term constructor over the differential operation because of the simplicity of its implementation in abstract machines.

Let us just point out that the use of \( D \) and of \( (\partial x.) \) are equivalent. Indeed, one writes \( \lambda x.(\partial x.t)u \) for \( D(\lambda x.t) \cdot u \). Conversely, one could use \( (D(\lambda x.t) \cdot u)x \) for \( (\partial x.t)u \).

3.1 A syntax enriched with a linear substitution

We describe a call-by-name differential λ-calculus, where the variable substituted in the linear substitution process is the first one to appear in (weak) head-position in the reduction process.

Definition 1. The terms of the differential call-by-name λ-calculus are:

\[
t, u, v ::= x \mid \lambda x.t \mid tu \mid (\partial x.t) \text{ with } x \text{ free in } t.
\]

In a linear substitution \( (\partial x.t) \) \( x \) is supposed to be free in \( t \) in order for the differential KAM to simulate the calculus. Indeed, the term \( u \) which will substituted in \( x \) will be remembered of in a global environment. We need \( x \mapsto u \) to be gone when the computation of \( t \) is over.

Definition 2. A variable is active in a term if it is free in this term and if it in the subterm under consideration for reduction,:

\[
\text{av}(x) = \{x\}, \text{av}(\lambda x.s) = \emptyset \text{ av}(st) = \text{av}(s), \text{av}((\partial x.s)) = \text{av}(s).
\]

The intuition is that \( x \) is active in \( s \) if and only if \( (\partial x.s)u \) will reduce.
Figure 2: The differential call-by-name \( \lambda \)-calculus

The calculus is typed by simple types enriched with a linear type borrowed from linear logic:

\[
A, B := \alpha | A \rightarrow B | A \rightarrow B.
\]

Terms are typed with the typing rules of simply-typed \( \lambda \)-calculus, enriched with:

\[
\begin{align*}
\Gamma, x : \alpha & \vdash t : B \\
\Gamma & \vdash (\partial x.t) : \alpha \Rightarrow B \\
\Gamma & \vdash u : A \\
\Gamma & \vdash tu : B
\end{align*}
\]

3.2 Weak and strong operational semantics

A differential weak head reduction In \((\partial x.t)u\) we choose to substitute only the weak head occurrence of \(x\) in \(t\) by \(u\). Thus, either \(t = xt'\) and then \((\partial x.t)u \rightarrow ut'\), or \(t\) can be reduced to \(t'\) in a call-by-name strategy and then:

\[
(\partial x.t)u \rightarrow (\partial x.t')u.
\]

We detail the reduction rules of the differential call-by-name \( \lambda \)-calculus in figure 3.2. To this calculus we associate a differential Krivine’s machine (figure 3.3), as detailed in section 3.3.

Observe that there is no critical pair for these reduction rules, and that it is thus a deterministic calculus.

The (weak) differential call-by-name \( \lambda \)-calculus would compute subterms of the sum in differential \( \lambda \)-calculus, in a syntax where no term would have an abstraction as normal form. Indeed, we don’t linearly substitute under an abstraction here, as we follow the reduction of terms with a weak strategy.

A strong call-by-name reduction So as to have a better correspondence with differential \( \lambda \)-calculus, we also detail the rules of a strong differential call-by-name \( \lambda \)-calculus. Indeed, differential \( \lambda \)-calculus allows linear substitution under the abstraction:

\[
(\partial x.\lambda y.s)T = \lambda y.(\partial x.s)T.
\]

The rules of strong differential call-by-name \( \lambda \)-calculus are the rules of the differential call-by-name \( \lambda \)-calculus, to which are added:

\[
\begin{align*}
\frac{s \rightarrow s'}{\lambda x.s \rightarrow \lambda x.s'}
\end{align*}
\]

where the rule of \( \beta \)-reduction is restricted to abstraction in normal form:

\[
(\lambda x.s)u \rightarrow_n s[u/x] \text{ when } s \text{ is in normal form},
\]

and where the notion of active variable for an abstraction is changed:

\[
av(\lambda x.s) = av(s).
\]
Theorem 3. Consider \( t \) a term of the differential \( \lambda \)-calculus such that \( t \to^* \sum_{i=1}^n t_i \) where the \( t_i \) are terms of \( \lambda \)-calculus. Then there is a (strong) differential call-by-name term \( t' \) and \( i \in \{1, \ldots, n\} \) such that \( t \to^* t' \to^*_{\pi} t_i \).

We require that the \( t_i \) are terms of \( \lambda \)-calculus so that to avoid differential application which would not reduce, such that \( Dx \cdot u \). Indeed, such terms do not translate into differential call-by-name \( \lambda \)-calculus.

The relation \( \to \) (resp. \( \to_{\pi} \)) represents the transitive closure of \( \to \) (resp. \( \to_{\pi} \)). Intuitively, if \( t \) is the term \( Ds \cdot u \) where \( s \) reduces to \( \lambda x. s' \), and if \( u \) reduces to \( u' \) then \( t' \) is the term \( \lambda x. (Dx.s')u' \). The term \( t' \) is written \( \lambda x.(Dx.s')u \) in differential call-by-name \( \lambda \)-calculus.

Proof. We prove the theorem by induction on \( t \). If \( t = x \) then no reduction step is needed. If \( t = \lambda x.s \) either \( t \) is in normal form and \( t = t'' \), or either \( s \to s' \to^* \sum s_i \), where \( t_i = \lambda x.s_i \). We apply then the induction hypothesis on \( s' \): there is \( s'' \) and \( j \) such that \( s' \to^* s'' \to^*_{\pi} s_j \). Then we prove the theorem by writing \( t' = \lambda x.s'' \) and \( t_j = \lambda x.s_j \). The case where \( t = tu \) is treated likewise, distinguishing the case where \( t \) is in normal form from the one where it is not.

Let us now suppose that \( t = Ds.u \). Either \( s \to s' \), and we can apply the induction hypothesis on \( s' \), or \( s \) is in normal form. In that case, as the \( t_i \)'s are terms of \( \lambda \)-calculus, \( s = \lambda x.s' \), where \( x \) is free in \( s' \) and \( s' \) is a term of \( \lambda \)-calculus in normal form. Thus \( t \to \lambda x.(Dx.s')u \), where \( (Dx.s')u = \sum s_i \) and \( t_i = \lambda x.s_i \).

Let us show by induction on \( s' \) that there is \( i \) such that in differential call-by-name \( \lambda \)-calculus \( (Dx.s')u \to_{\pi}^* s_i \). If \( s' = x \) this is immediate. Suppose that \( s' = s''u' \). Suppose that \( x \) is not free in \( s'' \) but is in \( u' \). Then in differential \( \lambda \)-calculus, \( (Dx.s''u')u = 0 + (Dss'' \cdot ((Dx.u')u)u') \). As \( s' = s''u \) is in normal form, \( s'' \) is not an abstraction and thus \( Dss'' \cdot ((Dx.u')u)u' \) does not reduce. We have a contradiction, as we supposed that all the terms appearing in the sum \( \sum t_i \) were terms of \( \lambda \)-calculus. Thus \( x \) must be free in \( s'' \).

Then there is by induction a term \( s''_j \) such that \( (Dx.s'')u \to_{\pi}^* s''_j \). As \( (Dx.s'')u = \sum s_i \), we have \( (Dx.s')u \to_{\pi}^* s_j \).

As \( s' \) is in normal form, \( s \) is not a differential application, and the only case left is \( s' = \lambda y.s'' \). We apply the induction hypothesis on \( s'' \) and make use of the rule \( \lambda \).

\[ \square \]

3.3 The differential Krivine’s machine

We introduce a differential version of the Krivine’s abstract machine (KAM) in figure 3.3. This is done by adding to the KAM a temporary and prioriety global environment stack, where terms in argument for a linear substitution are to be remembered until they are used.

The states of the differential KAM are quadruplets \( (x,e,\pi,T) \), in which:

- \( t \) is a term of the differential call-by-name calculus,
- \( e \) is the environment, represented as a partial function from the set of variables to the set of closures,
- a closure is a couple \( (t', e') \) of a term and an environment,
- \( \pi \) is the stack, represented by a finite sequence of closures,
- \( T \) is the temporary environment, represented as a partial function from the set of variables to the set of closures.

The rules \( \mathbb{S} \) and \( \mathbb{E} \) of the differential KAM correspond respectively to the linear substitution and to the substitution of a variable. In the linear substitution of \( x \) by \( t \), \( x \) is suppressed from the domain of \( T \) after being substituted. In the application rule \( \mathbb{2} \) only the global environment is captured in the closure, as we don’t want linearly substitute a variable twice. Then rule \( \mathbb{10} \) means that an abstraction takes a closure into the environment while rule \( \mathbb{11} \) means that the linear substitution takes a closure into the temporary environment.

Notice that the temporary environment \( T \) is never captured in a closure.

Definition 4. If \( T \) is a partial map from variables to closures, we are going to define \( \tilde{T}((t,e) : \pi) \) as the translation in differential call-by-name \( \lambda \)-calculus of the state \( (t,e,\pi,T) \). Informally, if \( \pi \) is a stack of closures \( (t_1, e_1) \vdash \ldots \vdash (t_n, e_n) \), then \( \tilde{T}(\pi) \) is the term \( t_1 \ldots t_n \) in which which variables are replaced by their value in \( T \) uppermost (and then suppressed in \( T \)) or in the corresponding \( e_i \). More precisely, \( \tilde{T} \) is defined inductively by the following rules:

- \( \tilde{T}((x, e) : \pi) = \tilde{T}^n(t, e') \tilde{T}(\pi) \) if \( T(x) = (t, e') \) and \( T' = T - (x, (t, e')) \).
Theorem 6. If \( t, \lambda \) call-by-name

Let us consider the term \( \lambda t.e, c : \pi, T \to t, e, \pi, T \) if \( T(x) = (t, e') \)

\( t, e, \pi, T \to t, e', \pi, T \) if \( T(x) = \emptyset \) and \( e(x) = (t, e') \)

\( \lambda x.t, e, c : \pi, T \to t, e + (x, c), \pi, T \)

\( \partial x.t, e, c : \pi, T \to t, e, \pi, T + (x, c) \)

\( tu, e, \pi, T \to t, e, (u, e) : \pi, T \)

\( \lambda x.t, e, c : \pi, T \to \emptyset, e + (x, c), \pi, T \)

\( \partial x.t, e, c : \pi, T \to \emptyset, e, \pi, T + (x, c) \)

\( tu, e, \pi, T \to \emptyset, e, (u, e) : \pi, T \)

(8) (9) (10) (11) (12)

Figure 3: The differential Krivine’s abstract machine

- \( \overline{T}((x, e) : \pi) = \overline{T}(t, e)\overline{T}(\pi) \) if \( T(x) = \emptyset \) and \( e(x) = (t, e') \).
- If \( t \) is not a variable, and \( T' \) is what is left of \( T \) when all the variables substituted in \( t \) from \( T \) have been suppressed, then \( \overline{T}((t, e) : a(t_1, e_1) : \pi) = \overline{T}((t, e))\overline{T}'((t_1, e_1) : \pi) \)

Proposition 5. \( \overline{T}((x, e) : \pi) \) is well-defined for every state \( t, e, \pi, T \) of the differential KAM.

Proof. \( \overline{T}((x, e) : \pi) \) is well-defined when \( \pi \) is empty, and the inductive definition of \( \overline{T}(\pi) \) uses \( \overline{T} \) on stack of strictly smaller size.

Theorem 6. If \( t \) is a differential call-by-name lambda-term, if in the differential KAM \( t, e, \pi, T \to t', e', \pi', T' \), then \( \overline{T}((t, e) : \pi) \to_n \overline{T}'((t', e') : \pi') \).

Proof. We prove this by induction on the term \( t \). Rules [9][10][12] of figure 3.2 do not involve \( T \), one can refer at [16] for the proof in these cases.

If rule [8] is involved in the differential KAM, it means that \( t = x, e = e', \pi = \pi', T(x) = (t', e') \) and \( T' = T - (x, (t', e')) \). Then \( \overline{T}((t, e) : \pi) = \overline{T}'((t', e') : \pi') = \overline{T}'((t', e') : \pi') \). No reduction rule is involved in differential call-by-name \( \lambda \)-calculus.

If rule [11] is involved in the differential KAM, it means that \( t = (\partial x.t), e = e', \pi = c : \pi' \) and \( T' = T + (x, c) \). Then as \( x \) is free in \( t' \), \( \overline{T}'((t', e') : \pi) \) will erase \( x \) from the domain of \( T' \), and \( \overline{T}'((t', e) : \pi) = \overline{T}'((t', e) : \pi') \). Since \( \overline{T}((\partial x.t', e) : c : \pi') \) reduces indeed in \( \overline{T}'((t', e : \pi') \), we have that \( \overline{T}((t, e) : \pi) \to_n \overline{T}'((t', e') : \pi') \).

As a consequence of this theorem, the differential KAM simulates indeed the differential call-by-name \( \lambda \)-calculus: if \( (t, \emptyset, \emptyset, \emptyset) \to^\ast (t', \emptyset, \emptyset, T) \) then up \( t \to_n \overline{T}(t, e) \). This theorem also shows that differential call-by-name \( \lambda \)-calculus indeed linearly substitute only the weak head occurrence of a variable.

An example of computation in the differential KAM Let us consider the term \( (\partial x.((\lambda z.z)x))y \).

The corresponding reduction in the CEK machine is the following:

\[
((\partial x.((\lambda z.z)x))y, \emptyset, \emptyset, \emptyset) \\
\to (\partial x.((\lambda z.z)x), \emptyset, (y, \emptyset), \emptyset) \\
\to ((\lambda z.z)x, \emptyset, (y, (x, (y, \emptyset)))) \\
\to (\lambda z.z, \emptyset, (x, (y, (x, (y, \emptyset)))) \\
\to (z, (z, (x, \emptyset)), (x, (y, \emptyset))) \\
\to (x, (z, (x, \emptyset)), (x, (y, \emptyset))) \\
\to (y, (y, (z, (x, \emptyset))), (x, (y, \emptyset))) \\
\to (y, (y, (y, (z, (x, \emptyset)))), (x, (y, \emptyset)))
\]

We have indeed \( (\partial x.((\lambda z.z)x))y \to_n y \).
4 A differential call-by-value \(\lambda\)-calculus

We describe a call-by-value differential \(\lambda\)-calculus, where the only variable to be linearly substituted would be the first one to be isolated from a context in a left-to-right call-by-value evaluation strategy. Meanwhile, usual terms are evaluated in a left-to-right call-by-value evaluation strategy. As in the call-by-name case, the \(D\) operator from differential \(\lambda\)-calculus is replaced by the linear substitution \(\partial\), because of its easiness of implementation in the CEK machine. We detail first a weak version of this calculus, which is computed by a differential version of the CEK machine, and a strong one which embeds into the differential \(\lambda\)-calculus.

4.1 A syntax enriched with linear substitution

Terms of the differential call-by-value \(\lambda\)-calculus are

\[ t, u ::= \text{x} \mid \lambda x.t \mid tu \mid (\partial x.t) \text{ where } x \text{ is free in } t, \]

while values are

\[ v ::= \text{x} \mid \lambda x.t \mid (\partial x.t) \text{ where } x \text{ is free in } t \]

As in the call-by-name case, we define active variables:

\[ \text{av}(x) = \{ x \}, \text{av}(\lambda x.s) = \emptyset, \text{av}(su) = \text{av}(s) \cup \text{av}(u), \text{av}((\partial x.s)) = \text{av}(s) \setminus \{ x \}. \]

This language is typed exactly as the call-by-name differential \(\lambda\)-calculus.

4.2 Weak and strong operational semantics

As in the call-by-name case, we want to linearly substitute the first encountered occurrence of the wanted variable. In this case however, the operational semantics is more intricate. Suppose we want to linearly substitute \(x\) by \(t\) in \(su\). As we are in a left-to-right strategy we will begin to compute \(s\). Suppose \(x\) does not appear during this computation. \(s\) will result in a value. If \(s\) is an abstraction \(\lambda z.s'\) as it should be in a typed system, we may proceed to the computation of \(u\). If \(x\) appears in the computation of \(u\), then we will linearly substitute it by \(t\). However, for \(t\) to be linear in \((\lambda z.s')u\) we need to linearize \(\lambda z.s'\). This corresponds to the second summand of rule 1, and is the computation interpretation of the chain rule as explained by Ehrhard and Regnier.

We write this rule in a continuation-passing style:

\[
\frac{}{(\partial x.v)v' \rightarrow_{\nu} v''} \quad (\partial x.\lambda z.s)v' \rightarrow_{\nu} (\lambda k.((\lambda z.k)v)((\partial s.z)v''))
\]

The rules of differential call-by-value \(\lambda\)-calculus are detailed in figure 4.2. This calculus is computed step-by-step in a differential CEK machine (figure 4.3), as shown in section 4.3.

Observe that there is no critical pair for these reduction rules, and that it is thus a deterministic calculus.

A strong call-by-value reduction

So as to have a better correspondence with differential \(\lambda\)-calculus, we also detail the rules of a strong differential call-by-value \(\lambda\)-calculus. These are the rules of the differential call-by-value \(\lambda\)-calculus, enriched with the following ones:

These are the rules of the differential call-by-name \(\lambda\)-calculus, enriched with:

\[
\frac{}{s \rightarrow s'} \quad \frac{}{\lambda x.s \rightarrow \lambda x.s'}
\]

where the rule of \(\beta\)-reduction is restricted to abstraction in normal form:

\[
(\lambda x.s)v \rightarrow_n s[v/x] \text{ when } s \text{ is in normal form,}
\]

and where the notion of active variable for an abstraction is changed:

\[
\text{av}(\lambda x.s) = \text{av}(s).
\]

The strong differential call-by-value \(\lambda\)-calculus computes one of the subterms of a differential \(\lambda\)-calculus term:
Theorem 7. Consider \( t \) a term of the differential \( \lambda \)-calculus such that \( t \rightarrow^* \sum_{i=1}^n t_i \) where the \( t_i \) are terms of \( \lambda \)-calculus. Then there is a (strong) differential call-by–value term \( t' \) and \( i \in \{1, \ldots, n\} \) such that \( t \rightarrow^* t' \rightarrow^* t_i \).

Proof. The proof is done as in the call-by-name case (Theorem 3). \( \square \)

4.3 The differential CEK machine

We detail a differential version of the CEK [13], without control operators (see figure 4.3). This is done by adding temporary global environment which is priority. It is global in the sense that it is never enclosed in a closure, as the usual local environment is. The implementation is slightly more complicated than for the KAM, as we must linearize a function whenever its argument bore a linear substitution. The differential version of the CEK with control operators should implement the differential \( \lambda \mu \)-calculus of Vaux [19].

The objects of the differential CEK machine are quadruplet \((t, e, \pi, T)\) in which:

- \( t \) is a term of the differential call-by–value-calculus,
- a closure \( c \) is a pair \((t, e)\) of a term and of an environment,
- \( e \) is an environment, represented as a partial map between variables and closures,
- \( \pi \) is the stack, represented as a list. Its elements either are arguments of a closure \( a(c) \) or functions of a closure \( f(c) \),
- \( T \) is the temporary environment, represented as a partial map between variables and closures.

The distinction between argument and functions in the CEK machine allows to implement to chain rule we talked about. Indeed, if \( x \) is to be substituted by a value in \( T \), and if the head of the stack is a function, then this function should be linearized. This corresponds to rule 14 of the machine.

Partial correctness of the CEK machine We prove the partial correctness of the differential CEK machine. Proving total correctness, as for the differential KAM, is work in progress.

Notation. When \( t \) is a term and \( e \) is an environment, we write \( t\{e\} \) the term \( t \) where variables are replaced by their value in \( e \).

Lemma 8. Let \( s \) be a term, \( x \) a variable free in \( s \) and \( v, v' \) values such that \((\partial x.s)v \rightarrow^* v'\). Then for any environment \( e \) and stack \( \pi \), we have in the CEK machine:

\[
\begin{align*}
  (s, x, \pi, (x, (v, v'))) & \rightarrow^* (v'(e'), e, \pi, \emptyset) \\
  \text{if } \pi \neq f(\lambda z.t', e') & \vdash \pi'
\end{align*}
\]
\[ x, e, \pi, T \rightarrow t, e', \pi, T - (x, (t, e')) \] if \( T(x) = (t, e') \) and \( e \neq f(\lambda z.t', e') \).

(13)

\[ x, e, f(\lambda z.t', e') :: \pi, T \rightarrow t, e'', f(\partial z.t', z = (x, e) + e') :: \pi, T - (x, (t, e'')) \] if \( T(x) = (t, e'') \).

(14)

\[ x, e, \pi, T \rightarrow t, e', \pi, T \] if \( T(x) = \emptyset \) and \( e(x) = (t, e') \).

(15)

\[ tu, e, \pi, T \rightarrow t, e', a(u, e) :: \pi, T \]

(16)

\[ v, e, a(u, e') :: \pi, T \rightarrow u, e', f(v, e) :: \pi, T \]

(17)

\[ v, e, f(\lambda x.t') :: \pi, T \rightarrow t, (x = (v, e)) + e', \pi, T \]

(18)

\[ v, e, f(\partial x.t', e') :: \pi, T \rightarrow t, e', \pi, (x = (v, e)) + T \]

(19)

Figure 5: The differential CEK machine.

\[
(s, e, f(\lambda z.t', e'') :: \pi, (x, (v, e'))) \rightarrow (u' \{e'\}, e, f(\partial z.t', z = (s, e) + e'') :: \pi, \emptyset).
\]

\[
\rightarrow
\]

\[
(s, e, (x, (v, e'))) \rightarrow
\]

\[
(u, e, f(\lambda z.t, e) :: \pi, (x, (v, e'))) \rightarrow
\]

\[
(u', e, f(\partial z.t, z = (u, e) + e) :: \pi, \emptyset) \rightarrow
\]

\[
(t', z = (u, e') + e), (z, (u', e')) \rightarrow
\]

\[
Let us write \((\partial z.t)u' \rightarrow u''\) where \(u''\) is a value. Then by induction hypothesis:
\[
(t', z = (u, e) + e), (z, (u', e')) \rightarrow
\]

\[
(u' \{e\}, z = (u, e) + e), (\pi, \emptyset) \rightarrow\]

\[
(v' \{e'\}, e, \pi, \emptyset)
\]

where \(\pi_0\) depends of the head of \(\pi\).

\[
\]

Let us consider now the case where \(s = (\partial z.t)u\) with \(x\) free in \(u\). Then if \((\partial z.u)u' \rightarrow v'\) where \(u'\) is a value, we have \((\partial z.t)u' \rightarrow v'\). Thus in the differential CEK machine:

\[
((\partial z.t)u, e, \pi, (x, (v, e))) \rightarrow
\]

\[
(u, e, f((\partial z.t), e) :: \pi, (x, (v, e))) \rightarrow
\]

\[
(u' \{e\}, e, f((\partial z.t), e) :: \pi, \emptyset) \rightarrow\]

\[
(t, e, (z, (u, e)) \rightarrow
\]

\[
(v' \{e'\}, e, \pi, \emptyset)
\]

The last derivation holds as \(u \{e\} = u(e)\). The other cases are treated likewise.

**Theorem 9.** Consider \(t\) a term, \(v\) a value and \(e\) an environment such that \(t \{e\} \rightarrow v(e)\). Then for any stack \(\pi\), we have in the CEK machine

\[
(t, e, \pi, \emptyset) \rightarrow (v, e, \pi, \emptyset).
\]

**Proof.** We prove this result by induction on \(t\) and case analysis on the first rule of \(t \{e\} \rightarrow v(e)\). We make use of the above lemma several times. \(\square\)
An example of computation for the CEK machine  Let us consider the term \((\partial x.((\lambda z.z)x))y\).
The corresponding reduction in the CEK machine is the following:

\[
\begin{align*}
((\partial x.((\lambda z.z)x))y, \varnothing, \varnothing, \varnothing) \\
\to ((\partial x.((\lambda z.z)x), \varnothing, (y, \varnothing), \varnothing) \\
\to ((\lambda z.z)x, \varnothing, (x, \varnothing, x \mapsto (y, \varnothing)) \\
\to (\lambda z.z, \varnothing, a(x, \varnothing), x \mapsto (y, \varnothing)) \\
\to (x, \varnothing, f(\lambda z.z, \varnothing), x \mapsto (y, \varnothing)) \\
\to (y, \varnothing, f(\partial z.z, z \mapsto (x, \varnothing), \varnothing) \\
\to (z, z \mapsto (x, \varnothing), \varnothing, z \mapsto (y, \varnothing)) \\
\to (y, z \mapsto (x, \varnothing), \varnothing, \varnothing)
\end{align*}
\]

In differential call-by-value \(\lambda\)-calculus, it reduces indeed to \(\lambda z.y\), which is \(\eta\)-equivalent to \(y\).

5 Differentiation of call-by-push value

We have just detailed a call-by-name and a call-by-value differential \(\lambda\)-calculus. One can merge these approaches by trying to differentiate a computational language in which both differential call-by-name and differential call-by-value lambda-calculi would embed.

We recall the terms of the call-by-push-value language. We include products, so as to make an example for differentiating composition, but not sums so as to keep the exposition concise.

\[
t, u ::= \lambda x.t | tv | \text{return } v \mid \text{let } x = t \text{ in } u \mid \text{force } v \mid \text{pm as } (x, y) \text{ in } t.
\]

\[
v ::= x \mid \text{thunk } t \mid (v, v')
\]

In the spirit of the enriched effect calculus, we add a differentiation operator \(D\) which operates on terms, and an asymmetric tensor product return \(v \otimes u\).

Computations of differential call-by-push-value are:

\[
t, u ::= \ldots \mid (D t) \mid \text{return } v \otimes u
\]

Values of differential call-by-push-value are values of call-by-push-value.

We recall the usual call-by-name and call-by-value translations of \(\lambda\)-calculus into call-by-push-value, and extend it to the differential call-by-name or call-by-value lambda-terms.

Definition 10. The call-by-name translation

\[
\begin{align*}
(x)^n &= \text{force } x \\
(\lambda x.t)^n &= \lambda x. t^n \\
(tu)^n &= t^n(\text{thunk } u^n) \\
(\partial x.t)^n &= \lambda u. \text{let } x = t \text{ return } x \otimes \text{force } u \text{ in } t^n
\end{align*}
\]

Definition 11. The call-by-value translation

\[
\begin{align*}
(x)^v &= \text{return } x \\
(\lambda x.t)^v &= \text{return thunk } \lambda x. t^n \\
(tu)^v &= \text{let } x = t \text{ in let } y = u \text{ in } (\text{force } x)y \\
(\partial x.t)^v &= \text{return } (\lambda u. \text{let } x = t \text{ return } x \otimes u \text{ in } t^v)
\end{align*}
\]

Typing We do not emphasize on typing in this article. The types are the ones of call-by-push-value enriched with the EEC asymmetrical tensor product computation type \(!A \otimes B\). The term \((D t)u\) has the type of \(t\), while the term \(v \otimes u\) has type \(!A \otimes B\) when the value \(v\) has type \(A\) and the computation \(u\) has type \(B\).
Differentiation and linear substitutions Two new group of reduction rules will be introduced. 
\((Dt)(\text{thunk } u)\) will replace the first subcomputation of \(t\) by \(u\). Thus we will need a reduction rule per computation constructor. The computation "let \(x = \text{return } v \otimes u\) in \(t'\) will replace \(x\) linearly by \(v'\) in \(t\), and then replace \(x\) non-linearly by \(v\). As when computing \((\partial x.t) v'\) in differential call-by-name or call-by–value \(\lambda\)-calculus, a reduction rule per computation constructor will be needed. It appears that \(D\) is more of a call-by-name term, while \(\text{return } v \otimes u\) is more used in a call-by–value point of view.

**Definition 12.** As in the differential call-by-name or call-by–value lambda-calculi, we define a notion of active variable in order to construct the reduction rules for linear substitution. We write \(V\) the set of all variables.

- \(av(x) = \{x\}\), \(av(tv) = av(t)\), \(av(\lambda x.t) = \emptyset\),
- \(av(\text{return } v) = V\), \(av(\text{let } x = t\ \text{in } u) = av(u)\),
- \(av(Dt) = av(t)\), \(av(\text{return } v \otimes u) = av(u)\)
- \(av(\text{force } v) = av(v)\), \(av(\text{thunk } t) = av(t)\),
- \(av(\text{pm } v\ \text{as } (x,y)\ \text{in } t) = av(t)\), \(av(v,v') = av(v) \cup av(v')\).

**The operational semantics of differential call-by-push-value** We explain a few rules of figure 5. The differential operator \(D\) substitute the first subcomputation of a computation. Thus:

\[
D(tv).u \rightarrow ((Dt)u)v,
\]

or also

\[
\text{let } x = t \text{ in } t' v \rightarrow \text{let } x = (Dt) v \text{ in } t'.
\]

The rules concerning \(\text{return } v \otimes u\) need to be understood in term of linear resources. For example, we have:

\[
\begin{align*}
\text{let } x &= \text{return } v \otimes u \text{ in } t' v' \\
&\rightarrow \text{let } x = \text{return } v \text{ in } t' v' \otimes \text{let } x = \text{return } v \otimes u \text{ in } v'.
\end{align*}
\]

This means that the result of a linear computation \((u \text{ in } t' v)\) will be a linear argument \(x = \text{return } v \otimes u\) in \(v'\). We defined \(av(\text{return } v) = V\) accordingly.

Another example of linear substitution rule is

\[
\begin{align*}
\text{let } x &= \text{return } v \otimes u \text{ in } t' t'' \\
&\rightarrow \text{let } x = \text{return } v \otimes u \text{ in } t \otimes t' \rightarrow \text{return } t \otimes t''.
\end{align*}
\]

In that case, we already have a place for the linear argument \(t''\). Thus \(t''\) will replace \(t'\) as a linear argument.

Linearity of computation and linearity of resources A term is said to be linear in a computation when it only needs to be computed once. For example, in a call-by–value point of view, \(u\) is linear in \((\lambda x.t)u\). However, in the traditional point of view of differential \(\lambda\)-calculus inspired from linear logic, \(u\) is not linear in \((\lambda x.t)u\) as it may be used more than once. This difference is illustrated by the reduction rule for \(D\text{return } v\): \(D\text{return } v\).u \rightarrow \text{return } v \otimes u.\)

**Theorem 13.** If \(t\) is a differential lambda-term, then if \(t \rightarrow_n t'\) then \((t)^n \rightarrow (t')^n\) in differential call-by-push-value. If \(t \rightarrow_v t'\) then \((t)^v \rightarrow (t')^v\) in differential call-by-push-value.

**Proof.** We detail the call-by-name case. The proof is done by induction on \(t\) and case analysis on the rule used for \(t \rightarrow_n t'\). The cases for \(\beta\)-reduction or head reduction are straightforward. If \(t = (\partial x.x)u \rightarrow t' = u\) then \(t^u = (\text{thunk } u) \rightarrow \text{let } x = \text{return } x \otimes u\ \text{in } x\) \(\rightarrow \text{let } x = \text{return } x \otimes u^u\ \text{in } x \rightarrow u^u\). If \(t = (\partial x.s)u\) and \(t' = (\partial x.s')u\) with \(x \notin av(s)\) and \(s \rightarrow_n s'\), then the reduction rule

\[
\begin{align*}
\text{let } x &= \text{return } v \otimes u \text{ in } t \rightarrow \text{let } x = \text{return } v \otimes u \text{ in } t'.
\end{align*}
\]
Usual reduction rules

\[(\lambda x.t)v \to t[x:=v]\]

\[\text{let } x = \text{return } v \text{ in } t \to t[x:=v]\]

\[\text{force (thunk } t \text{) } \to t\]

\[\text{pm } (v, v') \text{ as } (x, y) \text{int } \to t[v/x, v'/y]\]

Context rules

\[t \to t'\]

\[\text{let } x = t \text{ in } u \to \text{let } x = t' \text{ in } u\]

\[t \to t'\]

\[u \otimes t \to u \otimes t'\]

\[D t \to D t'\]

\[t \to t'\]

\[tv \to t'v\]

\[u \otimes t \to u' \otimes t\]

\[v \to v'\]

\[\text{pm } v \text{ as } (x, y) \text{ in } t \to \text{pm } v' \text{ as } (x, y) \text{int}\]

Reduction rules for \(D\)

\[D(\text{let } x = t \text{ in } t')v \to \text{let } x = (Dt)v \text{ in } t'\]

\[D(\text{force thunk } t) \to Dt\]

\[\text{let } x = \text{return } x \otimes u \text{ in } t \to t'\]

\[(D(\lambda x.t)(\text{thunk } u)) \to \lambda x.t'\]

\[D(\text{return } v.(\text{thunk } u)) \to \text{return } v \otimes u\]

\[\text{let } x = \text{return } v \otimes w \text{ in } t \to t'\]

\[\text{let } y = \text{return } v' \otimes w' \text{ in } t' \to t''\]

\[D(\text{pm } (v, v') \text{ as } (x, y) \text{ in } t)(\text{force (thunk } w, \text{thunk } w')) \to t''\]

\[(Dtv)v' \to ((Dt)v')v\]

Reduction rules for \(\text{return } v \otimes u\)

\[\text{let } x = \text{return } v \otimes u \text{ in } x \to u\]

\[\text{let } x = \text{return } v \otimes u \text{ in } t \to t'\]

\[\text{let } x = \text{return } v \otimes u \text{ in } tv' \to \text{let } x = \text{return } v \text{ in } t'v'\]

\[x \notin av(t) \text{ then } t \to t'\]

\[\text{let } x = \text{return } v \otimes u \text{ in } t \to \text{let } x = \text{return } v \otimes u \text{ in } t'\]

\[\text{let } x = \text{return } v \otimes u \text{ in } \text{return } v' \to (\text{let } x = \text{return } v \otimes u \text{ in } v') \otimes (\text{let } x = \text{return } v \text{ in } \text{return } v')\]

\[\text{let } x = \text{return } v \otimes u \text{ in } t \to t''\]

\[\text{let } x = \text{return } v \otimes u \text{ in } (\text{let } y = t \text{ in } t') \to \text{let } x = \text{return } v \text{ in } (\text{let } y = t'' \text{ in } t')\]

\[\text{let } x = \text{return } v \otimes u \text{ in } D t \to D(\text{let } x = \text{return } v \otimes u \text{ in } t)\]

\[\text{let } x = \text{return } v \otimes u \text{ in } t' \to t''\]

\[\text{let } x = \text{return } v \otimes u \text{ in } \text{return } t \otimes t' \to \text{return } t \otimes t''\]

\[\text{let } x = \text{return } v_0 \otimes u \text{ in } t \to t'\]

\[\text{let } x = \text{return } v_0 \otimes u \text{ in } \text{pm } (v, v') \text{ as } (z, y) \text{ in } t \to \text{pm } (w, w') \text{ as } (z, y) \text{ in } t'\]

Figure 6: The differential call-by-push-value calculus
allows to conclude. The only case left is when \( t = (\partial x.s)t^n \) and \( t' = s't'^n \) with \( (\partial x.s)u \rightarrow s' \). Then \( t^n \rightarrow t'^n \) thanks to reduction rule

\[
\begin{align*}
\text{let } x = \text{return } v \odot u \text{ in } t & \rightarrow t' \\
\text{let } x = \text{return } v \odot u \text{ in tv'} & \rightarrow \text{let } x = \text{return } v \text{ in } t'v'
\end{align*}
\]

6 Perspectives

The differential call-by-push-value presented here is minimal. Further research will develop differentiation for other computations types, and would precise the links between differentiation and effects.

Another line of research would be to generalize the approach used to differentiate the KAM and the CEK machine to differentiate other abstract machine. It seems worthwhile to enquire whether a universal method can be applied to differentiate an evaluation strategy. That is, how can we recover the rules of linear substitution given a deterministic set of rules for evaluation?

The semantical study of differential call-by-name or call-by–value calculus is not done here. Of course, we need to explore the links between the traditional models of differential \( \lambda \)-calculus \([8, 15]\) and models of call-by-push value or of EEC, enriched with a differentiation operator. It should have close links with investigation done by Ehrhard recently \([10]\).

Our calculus seems to have strong links with the Dialecta study by P.-M. Pédrot \([18]\), whose translation \((\_)^*\) (resp. \((\_)_*\)) bears similarities with \(D\) (resp. \((\partial x._*)\)). This should result in a deeper understanding of the computational content of differentiation.

Acknowledgment

I would like to thank T. Ehrhard, R. Mogelberg, P.-M. Pédrot, and T. Seiller for discussions on differential, effect, call-by-push-value calculi. I wish to thank to PLS group in ITU Copenhagen who welcomed me and where I wrote this article.

References


