パラメトリシティに基づくプログラム微積分

森畑 明昌

漸増計算とは，ある入力に対して一度出力を求めた後，入力が僅かに変化した場合に，前回の出力を用いて新しい入力に対する出力を高速に得る手法である．Cai ら（PLDI 2014）は，ラムダ式で記述されたプログラムを漸増計算を達成するプログラムへと変形する操作を提案し，これを「微分」と名付けた．この「微分」は数学的な微分との類似はあるものの，両者の関係性は不明であった．しかも，Cai らによる「微分」はラムダ式の構文に操作や証明が依存しており，一般化や他の手法との関連性の議論が困難であった．本発表ではまず，Cai らの「微分」を多相型のパラメトリシティの観点から再定式化する．これにより，プログラムの具体的な表現から離れ，型で表された抽象的な情報のみで議論を行うことができ，理論的な見通しが良くなる．さらに，この「微分」と全く同様の操作により，数学的な「微分」を得られること，具体的にはそれが Elliott の自動微分手法（ICFP 09）の一般化となることを示す．さらに，同様の方法で離散的「微分」や数学的「微分」の逆演算である「和分」「積分」を行う可能性について論じる．

When an input is slightly modified, incremental computing enables efficient calculation of the corresponding output by using the previous output. Cai et al. (PLDI 2014) proposed a program transformation that takes a programs in a lambda calculus and results in a program, called *derivative*, that achieves incremental computing. Though their derivatives are somewhat similar to the mathematical derivatives, their relationship was unclear. Moreover, their transformation and its proof were tightly connected to syntactic manipulations, and therefore, it was difficult to study its generalization and its relationships to other methods. This presentation reformulates the transformation by Cai et al. via *parametricity* of parametric polymorphic types. This reformulation enables us to abstractly discuss the properties based on types. Then, this presentation shows that the same approach can derive mathematical derivatives. In fact, the transformation turned out to be a generalization of the automatic differentiation method by Elliott (ICFP 2009). In addition, this presentation discusses possibilities of applying the same approach to summation and integration, which are the inverses of finite differentiation and continuous (i.e, mathematical) differentiation, respectively.

1 はじめに

漸増計算とは，計算 f に対する入力 x の計算結果 $y = f\ x$ が既に求まっている際に，入力が少しだけ変化して $x' = x \oplus dx$ となったとき，y を用いて $f\ x'$ の計算を効率よく行う手法である．ここで，dx は（通常小さな）変更，$\oplus$ はその変更の適用を表す演算子である．

漸増計算手法は大きく静的な手法（漸増計算変換，incrementalization）と動的な手法（漸増評価，incremental evaluation）に分けられる．静的な手法では，漸増計算を行わないプログラムを，漸増計算を達成するプログラムへと変換する．動的な手法では，プログラムを変換するのではなく，その解釈を変更することで漸増計算を実現する．

Cai らは [2] 単純型付きラムダ計算を対象とした静的な漸増計算手法を提案し，この手法を漸増的ラムダ計算（incrementalizing lambda calclus）と名付けた．漸増的ラムダ計算では，漸増変換を行うためのプログラムを得る「微分」と呼ばれるプログラム変換が定義されている．特に，関数 $f :: A \to B$ を「微分」することで，以下の等式を満たす関数 $f' :: A \to \Delta A \to \Delta B$ が得られるこれは，f の入力を dx だけ変化させた時の f の結果は，$f\ x$ と $f'\ x\ dx$ を組み合わせて得ら

Program Differentiation and Integration based on Parametricity.

Akimasa Morihata, 東京大学大学院総合文化研究科, Dept. of Arts and Sciences, the University of Tokyo.

れる，ということを意味する．なお，ΔA および ΔB はそれぞれ A および B に対する変更の型である．

$$f\ (x \oplus dx) = f\ x \oplus f'\ x\ dx$$

漸増的ラムダ計算は興味深い手法だが，実用・理論の両面から以下のような疑問が残っていた．

- 単純型付きラムダ計算だけでなく，実用的なプログラミング言語のプログラムに対しても同様の「微分」が定義できるか？
- 漸増的ラムダ計算の「微分」は数学的な「微分」とどのような関係にあるか？

まず，実用的な観点からは，単純型付きラムダ計算は再帰等が表現できないため不十分である．しかし，Cai らの変換や証明は構文に依存したものとなっており，再帰をはじめ様々なプログラミング言語の構成要素が自然に取り扱えるかは非自明であった．また，理論的な観点からは，漸増的ラムダ計算の「微分（以下これを数学的な微分と区別して**差分**と呼ぶ）」が本当に数学的な「微分」であると言えるのかは定かでない．もし差分が微分に繋がるのであれば，「**プログラム微分積分学**」のようなものに繋がる可能性があるため，この点は是非明らかにしたい．

以上のような状況をふまえ，本発表では以下の 3 点を示す．

まず，漸増的ラムダ計算の差分化を多相型のパラメトリシティの観点から定式化し直す．この再定式化によって差分化の操作で得られるプログラムが変わるわけではない．そのため，この再定式化自身は非常に大きな貢献というわけではない．しかし，これにより，証明の見通しが良くなり，実用的なプログラムへの一般化も簡単になる．実際，以降の議論はこの再定式化に基づいている．また，漸増的ラムダ計算の着想である「プリミティブ演算を差分化し，その結果を高階関数を介して伝播する」という点をより明確に表すことができる点も長所である．

次に，漸増的ラムダ計算の差分化と同様の手法で数学的な微分が得られることを示す．具体的には，プリミティブ演算が（差分化可能なだけでなく）微分可能である場合，それを高階関数で組み合わせたプログラムが差分化と全く同様にして微分できることを示す．この結果は，漸増的ラムダ計算の差分化を微分に結びつけるだけではなく，得られた微分手法が Elliott の前向き自動微分 [4] である点でも興味深い．

さらに，これら差分・微分の考え方が，その逆演算である和分・積分へ至る可能性について述べる．差分・微分が定義されたとしても，和分・積分をどう定義すべきかは自明には定まらない．本研究では，和分を「入力の変更過程によってもたらされる出力の変化の集約」との観点から定式化する．この観点は，Morihata [10] の漸増計算手法で入力の複雑な変更を扱うために用いられたものであり，漸増計算の観点からは自然なものである．そして，プリミティブ演算が和分・積分可能であれば，これを組み合わせて得られるプログラムも和分・積分可能であることを示す．

2 準備

本稿では原則として Haskell [11] を用いてプログラムを表現する．特に，式 e が型 τ を持つ状況を $e :: \tau$ と記述する．*fst* と *snd* はそれぞれ組の最初の要素と 2 つ目の要素を抽出する関数である．Haskell では 2 項演算子は通常カリー化形式として扱うが，本稿では反カリー化形式で，つまり組を引数とする関数として扱う．

本研究の議論は Cai らの漸増的ラムダ計算 [2] を基礎としている．漸増的ラムダ計算では，基本型とその操作が一定の性質を満たすことを仮定し，それらをラムダ計算によって組み合わせたプログラムについての性質を論じている．この状況を捉えるため，本研究では ML 系のプログラミング言語における**モジュール**を用いる．

モジュールは定数および関数の集まりからなる．以下，定数を定数関数と同一視し，「定数および関数」を単に「関数」と呼ぶことにする．特に，以下では特定の型 X の値を扱う関数群からなるモジュールに注目する．関数 $f_1 :: \tau_1(X), f_2 :: \tau_1(X), \ldots, f_n :: \tau_n(X)$ からなるモジュールを $\mathcal{M} = (f_1, f_2, \ldots, f_n)$，その**シグネチャ**を $\mathcal{S}(X) = [\tau_1(X), \tau_2(X), \ldots, \tau_n(X)]$ とし，$\mathcal{M} :: \mathcal{S}(X)$ と記述する．例えば，整数値リストに関するモジュール $\mathcal{M}_{List} = ([], (:), null, uncons)$ のシグネチャは $\mathcal{S}_{List}(X) = (X, (Int, X) \to X, X \to Maybe(Int, X))$ として $\mathcal{M}_{List} :: \mathcal{S}_{List}([Int])$ となる．

以下，モジュールは言語の一級市民とし，これを引数とする関数を考える．直感的にはモジュールを単に関数の組（ないしレコード）と捉えれば良い．例えば $\mathcal{M}_{List}$ を引数とする形で，リストの要素順序を逆転する $reverse$ 関数を以下のように定義できる．

$$reverse :: \forall X.\ S_{List}(X) \to X \to Int$$
$$reverse\ (e, c, unc)\ x = aux\ x\ e$$
$$\textbf{where}\ aux\ x\ h = \textbf{case}\ unc\ x\ \textbf{of}$$
$$Nothing \to h$$
$$Just\ a\ y \to aux\ y\ (c\ (a, h))$$

モジュールを引数とすることで，リスト以外であっても同様に列をなすような値であれば反転を求められるような定義となっていることに注意せよ．

3 漸増的ラムダ計算

漸増的ラムダ計算では，モジュールの各関数が漸増計算に適している場合に，そのモジュールを組み合わせて記述したプログラムが漸増計算を達成することを示している．本稿では，各モジュールについて以下の性質を要求する [†1]．

まず，基本型 A に対し，その局所的な変化の型 ΔA および変化を適用する演算子 $(\oplus_A) :: (A, \Delta A) \to A$ が存在するとする．なお，以下の議論では簡単のため，$x \oplus_A \mathbf{0}_A = x$ を満たす変換の単位元 $\mathbf{0}_A :: \Delta A$ の存在を仮定する．誤解のない範囲で添字は省略する．ΔA はあくまで局所的な変化を表し，より一般的な変化は $A \to A$ の関数として扱う．例えば，リスト型 $[A]$ に対する $\Delta[A]$ は先頭要素の追加・削除のみを含むかもしれない．加えて，$a \in A$ に対して適用可能な局所的な変更の部分集合 $U_a \subseteq \Delta A$ を a の**近傍**と呼ぶ．なお，具体的にどのような変化を局所的だと捉えるかは応用によるため，本研究では定義の具体的な内実には深入りしない．

定義 3.1. 関数 $f :: A \to B$ [†2] が差分化可能であるとは，以下を満たす関数 $f_{\text{FD}} :: (A, \Delta A) \to \Delta B$（これを f の差分関数と呼ぶ）が存在することである．

$$f\ (x \oplus dx) = f\ x \oplus f_{\text{FD}}\ (x, dx)$$

Cai ら [2] は，モジュールの各関数が差分化可能であるとき，そのモジュールを用いて記述したプログラムに対し，その差分化関数を導出するプログラム変換を与えている．このプログラム変換の対象は不動点演算子を含まない単純型付きラムダ式に制限されていたが，Giarrusso ら [5] はこれを一般のラムダ式に拡張している．いずれの変換も，ラムダ式に対する構文的な変換として定式化されている．そのため，プログラム全体の変換が必要となるためライブラリとしての実装が困難である，言語の構成要素を増やした際の再証明が面倒である，等の問題点がった．本研究ではこのプログラム変換をパラメトリシティを用いることで再定式化する．

まず，モジュール $\mathcal{M} = (f_1, \ldots, f_n) :: \mathcal{S}(A)$ に対し，**差分付加モジュール** $\hat{\mathcal{M}}_{\text{FD}} = (\hat{f}_{1\text{FD}}, \ldots, \hat{f}_{n\text{FD}}) :: \mathcal{S}(\hat{A}_{\text{FD}})$（ただし $\hat{A}_{\text{FD}} = (A, \Delta A)$）を以下の通り定義する．

$$\hat{f}_{i\text{FD}}\ (x, dx) = (f_i\ x, f_{i\text{FD}}\ (x, dx))$$

このとき，差分関数導出定理は以下となる．

定理 3.2. モジュール $\mathcal{M} :: \mathcal{S}(A)$ とその差分付加モジュール $\hat{\mathcal{M}}_{\text{FD}} :: \mathcal{S}(\hat{A}_{\text{FD}})$，そしてパラメトリック多相型の関数 $g :: \forall X.\ S(X) \to X \to X$ について，以下が成り立つ．$g\ \hat{\mathcal{M}}_{\text{FD}}\ (x, dx) = (y, dy)$ であるとき，

- $g\ \mathcal{M}\ x = y$
- $g\ \hat{\mathcal{M}}_{\text{FD}}\ (x \oplus dx) = y \oplus dy$.

証明．パラメトリック多相型に関するパラメトリシティ [12] [15] により証明を行う．まず，$g\ \mathcal{M}\ x = y$ は $fst \circ g\ \hat{\mathcal{M}}_{\text{FD}} = g\ \mathcal{M}$ と等価であるが，これは各 $f_i \in \mathcal{M}$ について定義より明らかに $fst \circ \hat{f}_{i\text{FD}} = f_i$ であることより導かれる．また，$g\ \hat{\mathcal{M}}_{\text{FD}}\ (x \oplus dx) = y \oplus dy$ は $(\oplus) \circ g\ \hat{\mathcal{M}}_{\text{FD}} = g\ \mathcal{M} \circ (\oplus)$ と等価であるが，これは，各 $f_i \in \mathcal{M}$ が差分化可能であること，すなわち $(\oplus) \circ \hat{f}_{i\text{FD}} = f_i \circ (\oplus)$ を満たすことより導かれる． □

定理 3.2 は，$g\ \mathcal{M}$ の差分関数が $snd \circ g\ \hat{\mathcal{M}}_{\text{FD}}$ であることを述べている．すなわち，型 A の値を計算する関数について，A に関する計算を全て差分付加することで差分関数が得られることになる．これは実際

†1 これは Cai らの定式化よりはやや弱い条件だが，以下に見るように我々の目的には十分である．

†2 この関数を含め，以下現れる関数軍についての議論は適当な関手 F および G を用いて $f :: \mathsf{F}A \to \mathsf{G}A$ という型の関数の場合に自然に拡張できる．この議論は直裁なので本稿では深入りしない．

に既存研究[2][5]でのプログラム変換の骨子である．

4 高階差分

関係データベース処理などの文脈では高階差分，すなわち差分関数の差分など，が有用であることが知られている[8]．高階差分も全く同様の手法で導出できる．

いま，基本型 A に対し，その k 階までの変化の型 $\Delta^i A$（$0 \leq i \leq k$，$\Delta^0 A = A$，$\Delta^{i+1} A = \Delta(\Delta^i A)$ とする）および変化を適用する演算子 $(\oplus^i) :: (\Delta^{i-1} A, \Delta^i A) \to \Delta^{i-1} A$（$1 \leq i \leq k$）が存在するとする．関数 $f :: A \to B$ が k **階差分化可能**であるとは，全ての $1 \leq i \leq k$ について，以下を満たす関数 $f^{(i)}{}_{\text{FD}} :: (\Delta^{i-1} A, \Delta^i A) \to \Delta^{i-1} B$（これを f の i **階差分関数**と呼ぶ）が存在することである．

$$f^{(0)}{}_{\text{FD}} = f$$

$$f^{(i-1)}{}_{\text{FD}}\ (x \oplus dx) = f^{(i-1)}{}_{\text{FD}}\ x \oplus f^{(i)}{}_{\text{FD}}\ (x, dx)$$

この場合についても，モジュール $\mathcal{M} = (f_1, \ldots, f_n) :: \mathcal{S}(A)$ に対する，k **階差分付加モジュール** $\hat{\mathcal{M}^{(k)}}_{\text{FD}} = (\hat{f_1}_{\text{FD}}, \ldots, \hat{f_n}_{\text{FD}}) :: \mathcal{S}(\hat{A^{(k)}}_{\text{FD}})$（ただし $\hat{A^{(k)}}_{\text{FD}} = (A, \Delta A, \ldots, \Delta^k A)$）を以下のように定義する．

$$\hat{f}_{i\text{FD}}\ (x_0, x_1, \ldots, x_k) = (f_i\ x_0, f_i^{(1)}{}_{\text{FD}}\ (x_0, x_1), \ldots, f_i^{(k)}{}_{\text{FD}}\ (x_{k-1}, x_k))$$

このとき以下の定理が成り立つ．これは定理 3.2 の一般化であるが，証明は全く同様である．

定理 4.1. モジュール $\mathcal{M} :: \mathcal{S}(A)$ とその k 階差分付加モジュール $\hat{\mathcal{M}^{(k)}}_{\text{FD}} :: \mathcal{S}(\hat{A^{(k)}}_{\text{FD}})$，そしてパラメトリック多相型の関数 $g :: \forall X.\ S(X) \to X \to X$ について，以下が成り立つ．$g\ \hat{\mathcal{M}^{(k)}}_{\text{FD}}\ (x_0, x_1, \ldots, x_k) = (y_0, y_1, \ldots, y_k)$ であるとき，

- $g\ \mathcal{M}\ x_0 = x_1$
- $g\ \hat{\mathcal{M}}_{\text{FD}}\ (x_i \oplus x_{i+1}) = y_i \oplus y_{i+1}$．

定理 4.1 は，$g\ \mathcal{M}$ の i 階差分関数が $g\ \hat{\mathcal{M}}_{\text{FD}}$ の i 番目（ただし最初の要素を 0 番目とする）の要素の計算に対応することを述べている．

5 微分

ここまでの議論は，漸増計算を前提として，入力の変更を出力へ伝播することを考えてきた．特に入力の変更がごく僅かな場合の極限にどうなるかを論じるのが数学での微分である．我々の導出が微分にまでスケールするかは興味のあるところである．

関数の微分を議論するためには，その関数の定義域及び値域が一定の性質を満たさなければならない．本研究では，関数 $f :: A \to B$ について以下の状況を考える．

まず，型 A（ないし B）の値に対する変更 ΔA が，$(\oplus)$ に加え，距離関数 $d :: \Delta A \times \Delta A \to \mathbb{R}$ をもつものとする．略記法として $||dx|| = d(dx, \mathbf{0})$ とし，これを dx の大きさと呼ぶ．これらは「変更を小さくした際の極限」を議論する際に用いる．さらに，微分値の型を $(A \to B)_{\text{D}} \subseteq \Delta A \to \Delta B$ とする．$(A \to B)_{\text{D}}$ は以下の性質を満たすことをを要求する．

- $(A \to B)_{\text{D}}$ が関数合成に関して閉じていること，すなわち任意の $f \in (A \to B)_{\text{D}}$ および $g \in (B \to C)_{\text{D}}$ について $g \circ f \in (A \to C)_{\text{D}}$．
- 変更を顕著に増幅しないこと．すなわち任意の $f_{\text{D}} \in (A \to B)_{\text{D}}$，$a \in A$，および十分小さな a の近傍 U_a について，$\sup_{da_1, da_2 \in U_a} \frac{d(f_{\text{D}}(a \oplus da_1), f_{\text{D}}(a \oplus da))}{d(da_1, da_2)}$ が有限の値を持つ．

定義 5.1. 差分化可能関数 $f :: A \to B$ が入力 x で微分値 $f'_x :: (A \to B)_{\text{D}}$ をもつとは [†3]，

$$\lim_{||dx|| \to 0} \frac{d(f_{\text{FD}}\ (x, dx), f'_x\ dx)}{||dx||} = 0$$

を満たすことである．ここで f_{FD} は f の差分関数である．関数 f が入力 x で微分可能であるとは，関数 f が入力 x での微分値をもつことであり，関数 f が微分可能であるとは，f の定義域のすべての入力で微分値を持つこととする．関数 f が微分可能であるとき，f の微分 $f_{\text{D}} :: A \to (A \to B)_{\text{D}}$ とは $f_{\text{D}}\ x$ が f の x での微分値となるような関数である．

上記定義は，引き算の代わりに距離関数を用いている点を除き，標準的な微分の定義である．ただし，この定式化では，$(A \to B)_{\text{D}}$ は線形写像の集合とは限らない．微分値が実際に満たすべき制約（例えば線形性）は目的や型 A に依存すると思われるためここで

†3 $||dx|| \to 0$ を 0 に近づける方向が問題となる場合は十分考えられるが，本稿ではそれに踏み込まない．適切な定式化のもとで方向微分を考えるのは自然である．

は深入りしない．

差分の場合と同様，微分可能な関数のみからなるモジュールで書かれたプログラムは微分可能である．まず，モジュール $\mathcal{M} = (f_1, \ldots, f_n) :: \mathcal{S}(A)$ に対し，**微分付加モジュール** $\mathcal{M}_{\mathrm{D}} = (\hat{f}_{1_{\mathrm{D}}}, \ldots, \hat{f}_{n_{\mathrm{D}}}) :: \mathcal{S}(\hat{A}_{\mathrm{D}})$（ただし $\hat{A}_{\mathrm{D}} = (A, (A \to A)_{\mathrm{D}})$）を以下の通り定義する．ここで $f_{i_{\mathrm{D}}}$ は f_i の微分である．

$$\hat{f}_{i_{\mathrm{D}}}\ (x, f') = (f_i\ x, f_{i_{\mathrm{D}}}\ x \circ f')$$

このとき，以下の定理が微分を導出する．

定理 5.2. モジュール $\mathcal{M} :: \mathcal{S}(A)$ とその微分付加モジュール $\hat{\mathcal{M}}_{\mathrm{D}} :: \mathcal{S}(\hat{A}_{\mathrm{D}})$，そしてパラメトリック多相型の関数 $g :: \forall X.\ S(X) \to X \to X$ について，以下が成り立つ．$g\ \hat{\mathcal{M}}_{\mathrm{D}}\ (x, id) = (y, g'_x)$ であるとき，g'_x は $g\ \mathcal{M}$ の x での微分値である．

証明. パラメトリック多相型に関するパラメトリシティ [12] [15] により，各 f_i について，「f' が点 y 上での変化 dy をもたらす計算に対する適切な微分値であるとき，$f_{i_{\mathrm{D}}}\ y \circ f'$ も適切な微分値となる」ことを示せばよい．つまり，適切に dx を小さく取ることにより，$d(f_{i_{\mathrm{FD}}}(dy), (f_{i_{\mathrm{D}}}\ y \circ f')\ dx)$ が十分に（$||dx||$ に比例する程度以上に）小さくなることを示せばよい．

以下，証明の概要を述べる．正確な議論には標準的な $\epsilon - \delta$ 論法を使えばよい．d が距離関数であることを用いて以下のように式変形する

$$\begin{aligned}
& \frac{d(f_{i_{\mathrm{FD}}}\ dy, (f_{i_{\mathrm{D}}}\ y \circ f')\ dx)}{||dx||} \\
\leq\ & \frac{d(f_{i_{\mathrm{FD}}}\ dy, f_{i_{\mathrm{D}}}\ y\ dy)}{||dx||} + \\
& \frac{d(f_{i_{\mathrm{D}}}\ y\ dy, (f_{i_{\mathrm{D}}}\ y \circ f')\ dx)}{||dx||} \\
=\ & \frac{d(f_{i_{\mathrm{FD}}}(dy), f_{i_{\mathrm{D}}}\ y\ dy)}{||dy||} \frac{||dy||}{||dx||} + \\
& \frac{d(f_{i_{\mathrm{D}}}\ y\ dy, f_{i_{\mathrm{D}}}\ y\ (f'\ dx))}{d(dy, f'\ dx)} \frac{d(dy, f'\ dx)}{||dx||} \\
\leq\ & \frac{d(f_{i_{\mathrm{FD}}}(dy), f_{i_{\mathrm{D}}}\ y\ dy)}{||dy||} \frac{||f'\ dx|| + d(dy, f'\ dx)}{||dx||} + \\
& \frac{d(f_{i_{\mathrm{D}}}\ y\ dy, f_{i_{\mathrm{D}}}\ y\ (f'\ dx))}{d(dy, f'\ dx)} \frac{d(dy, f'\ dx)}{||dx||}
\end{aligned}$$

dx が $\mathbf{0}$ に近づく時，$\frac{d(f_{i_{\mathrm{FD}}}(dy), f_{i_{\mathrm{D}}}\ y\ dy)}{||dy||}$ は f_i の微分可能性から，$\frac{d(dy, f'\ dx)}{||dx||}$ は f' が微分値であることから，それぞれ 0 に近づく．一方で，$\frac{||f'\ dx||}{||dx||}$ および $\frac{d(f_{i_{\mathrm{D}}}\ y\ dy, f_{i_{\mathrm{D}}}\ y\ (f'\ dx))}{d(dy, f'\ dx)}$ は微分値の性質により有限値である．よって全体としては確かに 0 に漸近する． □

定理 5.2 は定理 3.2 同様微分値を付加したモジュールを用いることで系統的にプログラムの微分を得ることができることを示している．実は，この結果は Elliott による前向き自動微分手法 [4] の一般化となっている．定式化上は，Elliott が型クラスを用いているのに対し，本研究ではモジュールとパラメトリシティを用いてる点に差異があるが，得られる微分は同一である．

6 和分

差分・微分が適切に定義できた以上，それらの逆にあたる和分・積分を定義したくなるのは自然な要求である．しかし，和分・積分の定義はそれほど自明ではない．

数学的に採用される積分の標準的な定義の一つは「微分の逆演算」としての定義である．これに基づけば，関数 $f_{\mathrm{FD}} :: (A, \Delta A) \to \Delta B$ が与えられたとき，

$$f\ (x \oplus dx) = f\ x \oplus f_{\mathrm{FD}}\ (x, dx)$$

を満たす関数 $f :: A \to B$ を求めるのが和分の操作となる．しかし，よく知られているように，この定義を満たす積分は一意には定まらない．通常の微積分学の文脈では，積分定数を用いてこの問題を解決するが，プログラムの微積分でも同じような処理が可能であるかは定かでない．

もう一つの標準的な定義は，いわゆる定積分を基本とし，面積として積分を定義するものである．本稿でもこの方針を採用する．すなわち，関数 $f :: A \to B$ が与えられた時，$a_1, a_2 \in A$ に対して，

$$\int_{a_1}^{a_2} f \mathrm{d}x$$

を「f の引数を a_1 から a_2 まで変化させた時の f の値の総和」と捉えるものである．この定義を成立させるためには，以下の 2 点を解決する必要がある．

- 型 B 上で「総和」という演算が定義されていなければならない．
- 「a_1 から a_2 まで変化させる」方法が定義されていなければならない．

前者については，$\int_{a_1}^{a_2} f \mathrm{d}x$ の結果を B ではなく B に対する変更と取ることで，変更の累積を「総和」と

して用いる．この定義は以下の 3 点の理由から自然である．

- 定積分 $\int_a^b f(x)\mathrm{d}x$ を原始関数の値の差 $F(b) - F(a)$ と定義するのは自然である．型 B の値同士の差であるから，B に対する変更だと捉えるのは自然である．
- 積分結果に B 上での初期値を与えることで B の値を得ることができる．この初期値を積分定数と捉えれば，積分の曖昧性の自然な表現だと理解できる．
- Morihata [10] は，入力が非自明に変化した際の漸増計算を考え，入力の変化に対する出力の変化を累積する手法を示している．これは本稿で考える「積分」と同じ処理であり,「積分」の実用上の有用性を示唆する．

また，後者の問題については，目的ごとに用いるべき手法は異なると考えられるので，各具体的な演算ごとに規定されるものとする．

以上をふまえ，以下にまずは和分の定義を与える．

定義 6.1. 関数 $f :: (A, \Delta A) \to \Delta B$，値 $a_1, a_2 :: A$，に対し，和分 $\sum_{a_1}^{a_2} f :: B \to B$ は以下を満たす値である．

- $\sum_a^a f = \lambda x.\ x$
- $\sum_{a_1}^{a_2 \oplus da} f = \lambda x.\ (\sum_{a_1}^{a_2} f)\ x \oplus f\ (a_2, da)$

和分結果は局所的な変更とは限らないことに注意せよ．以下の定理が示すように，和分は差分の逆演算にほぼ対応する．

定理 6.2. f_{FD} が f の差分関数ならば，

$$(\sum_a^{a \oplus da} f_{\text{FD}})\ (f\ a) = f\ (a \oplus da)$$

である．

証明.

$$\begin{aligned} & (\textstyle\sum_a^{a \oplus da} f_{\text{FD}})\ (f\ a) \\ = & \quad \{\ \text{和分の定義}\ \} \\ & (\lambda x.\ x \oplus f_{\text{FD}}(a, da))\ (f\ a) \\ = & \quad \{\ \text{差分関数の定義}\ \} \\ & f\ (a \oplus da) \end{aligned}$$

□

定理 6.3. $f :: (A, \Delta A) \to \Delta B$ に対し，$g_b\ x = (\sum_a^x f)\ b :: A \to B$ とする．このとき，

$$g_b\ (x \oplus dx) = g_b\ x \oplus f\ (x, dx)$$

である．

証明. 和分の定義により直截である． □

定理 6.2 は差分関数の和分は元の関数とほぼ一致することを述べている．また定理 6.3 は，本質的には和分の差分関数が元の関数となることを示している．

差分や微分の場合と同様，和分可能な関数のみからなるモジュールで書かれたプログラムは和分可能である．まず，モジュール $\mathcal{M} = (f_1, \ldots, f_n, (\oplus)) :: \mathcal{S}(A, \Delta A)$ に対し，**和分付加モジュール** $\mathcal{M}_{\text{SUM}} = (\hat{f}_{1\text{SUM}}, \ldots, \hat{f}_{n\text{SUM}}, \lambda x, f.\ f\ x) :: \mathcal{S}(A, A \to A)$ を以下の通り定義する．

$$\hat{f}_{i\text{SUM}}\ (x, f') = \sum_x^{f'\ x} f_i$$

和分可能関数は変更 ΔA を本質的に計算に用いるため，変更の適用 $(\oplus)$ がモジュールに含まれていなければならないことに注意せよ．このとき，以下の定理が和分を導出する．

定理 6.4. モジュール $\mathcal{M} :: \mathcal{S}(A, \Delta A)$ とその和分付加モジュール $\mathcal{M}_{\text{SUM}} :: \mathcal{S}(A, A \to A)$，そしてパラメトリック多相型の関数 $g :: \forall X, Y.\ S(X, Y) \to (X, Y) \to Y$ について，$g\ \mathcal{M}_{\text{SUM}}\ (a_1, \lambda x.\ a_2) = \sum_{a_1}^{a_2} g\ \mathcal{M}$ である．

証明. パラメトリック多相型に関するパラメトリシティ [12][15] により，和分の性質を満たすことを示す．$g\ \mathcal{M}_{\text{SUM}}\ (a, \lambda x.\ a) = \lambda x.\ x$ は，各 f_i についての和分の性質から明らかである．よって，$g\ \mathcal{M}_{\text{SUM}}\ (a_1, \lambda x.\ a_2 \oplus da) = \lambda x.\ g\ \mathcal{M}_{\text{SUM}}\ (a_1, \lambda x.\ a_2)\ x \oplus g\ \mathcal{M}_{\text{SUM}}\ (a_2, da)$ を示せばよい．このためには各 f_i について，$f'\ x = y \oplus dy$ ならば $f_{i\text{SUM}}\ (x, f') = \lambda z.\ f_{i\text{SUM}}\ (x, \lambda x.\ y)\ z \oplus f_i\ (y, dy)$ が十分である．これも f_i の和分の性質から従う． □

7 積分

積分も和分と同様に定義できる．5 節と同様，変更 ΔA 上に距離関数 d が定義されているとする．さらに，積分を極限として議論するため，変化が小さい時の積分値についての距離を考えたい．これを可能とするため,「十分に小さな変化を行う関数 $f :: A \to A$」

については，それが局所的な変更と同一視できること，つまり $f = \lambda x.\ x \oplus da$ なる $da :: \Delta A$ が存在すること，を仮定し，定義域をそのような関数の上に拡張する．すなわち，上述のような f については，$d(f, dx) = d(da, dx)$ とする．

定義 7.1. 関数 $f :: (A, \Delta A) \to \Delta B$ が区間 $a_1, a_2 :: A$ で積分値 $\int_{a_1}^{a_2} f :: B \to B$ をもつとは以下の条件を満たすことである．

- $\int_a^a f = \lambda x.\ x$
- $\int_a^c f = (\int_b^c f) \circ (\int_a^b f)$
- $\lim_{||da|| \to 0} \frac{d(\int_a^{a \oplus da} f, f(a, da))}{||da||} = 0.$
- a の十分小さな近傍 U_a において，$\sup_{da_1, da_2 \in U_a} \frac{d(\int_a^{a \oplus da_1} f, \int_a^{a \oplus da_2} f)}{d(da_1, da_2)}$ が有限の値をもつ．

関数 f が定義域上の任意の区間で積分値を持つとき，これを積分可能であると呼ぶ．

1 つめと 2 つめの条件は，積分が和分とほぼ同等の性質を満たすことの表明である．積分は和分と違い，被積分関数が積分結果の正確な変化を記述しない．ただし，十分変化を小さく取れば，被積分関数が積分結果の変化に漸近することが期待される．これを述べているのが 3 つ目の条件である．4 つめの条件は，被積分関数が発散しないことを要求しており，これも自然である．

和分と同様，積分可能な関数のみからなるモジュールで書かれたプログラムは積分可能である．モジュール $\mathcal{M} = (f_1, \ldots, f_n, (\oplus)) :: \mathcal{S}(A, \Delta A)$ に対し，**積分付加モジュール** $\mathcal{M}_{\text{INT}} = (\hat{f}_{1_{\text{INT}}}, \ldots, \hat{f}_{n_{\text{INT}}}, \lambda x, f.\ f\ x) :: \mathcal{S}(A, A \to A)$ を以下の通り定義する．

$$\hat{f}_{i_{\text{INT}}}\ (x, f') = \int_x^{f'\ x} f_i$$

このとき，以下の定理が積分を導出する．

定理 7.2. モジュール $\mathcal{M} :: \mathcal{S}(A, \Delta A)$ とその積分付加モジュール $\mathcal{M}_{\text{INT}} :: \mathcal{S}(A, A \to A)$，そしてパラメトリック多相型の関数 $g :: \forall X, Y.\ S(X, Y) \to (X, Y) \to Y$ について，$g\ \mathcal{M}_{\text{INT}}\ (a_1, \lambda x.\ a_2) = \int_{a_1}^{a_2} g\ \mathcal{M}$ である．

証明．パラメトリック多相型に関するパラメトリシティ [12] [15] により，積分の性質を満たすことを示す．1 番目の性質は各 f_i の積分の性質により明らかである．2 番目の性質も，各 f_i の積分の性質，およびこれによって得られた関数合成が続く関数の入力を 2 区間に適切に分解することにより明らかである．4 番目の性質も各 f_i が上に有界であることより明らかに従う．

3 番目の性質を示すためには，各 f_i について「$f' = \lambda y.\ y \oplus dy'$ が x から $x \oplus dx$ までの和分によってもたらされるはずだった y からの変化 dy を積分により見積もったものだとして，$\hat{f}_{i_{\text{INT}}}\ (y, f')$ が適切な積分となる」ことを示せばよい．つまり，適切に dx を小さく取ることにより，真の和分を dz として $d(\int_y^{f'\ y} f_i, dz)$ が十分に（$||dx||$ に比例する程度以上に）小さくなることを示せばよい．

以下，証明の概要を述べる．正確な議論には標準的な $\epsilon - \delta$ 論法を使えばよい．d が距離関数であることを用いて以下のように式変形する．

$$\begin{aligned}
& \frac{d(\int_y^{f'\ y} f_i, dz)}{||dx||} \\
\leq\ & \frac{d(\int_y^{f'\ y} f_i, \int_y^{y \oplus dy} f_i) + d(\int_y^{y \oplus dy} f_i, dz)}{||dx||} \\
=\ & \frac{d(\int_y^{f'\ y} f_i, \int_y^{y \oplus dy} f_i)}{d(f', dy)} \frac{d(f', dy)}{||dx||} + \\
& \frac{d(\int_y^{y \oplus dy} f_i, dz)}{||dy||} \frac{||dy||}{||dx||} \\
\leq\ & \frac{d(\int_y^{f'\ y} f_i, \int_y^{y \oplus dy} f_i)}{d(f', dy)} \frac{d(f', dy)}{||dx||} + \\
& \frac{d(\int_y^{y \oplus dy} f_i, dz)}{||dy||} \frac{||f'|| + d(f', dy)}{||dx||}
\end{aligned}$$

ここで，仮定より $\frac{d(f', dy)}{||dx||}$ および $\frac{d(\int_y^{y \oplus dy} f_i, dz)}{||dy||}$ は 0 に漸近し，$\frac{d(\int_y^{f'\ y} f_i, \int_y^{y \oplus dy} f_i)}{d(f', dy)}$ と $\frac{||f'||}{||dx||}$ は有限なので，全体としても 0 に漸近する． □

なお，残念ながら本稿で定義した積分は本稿で定義した微分の逆演算ではない．微分は $(A \to B)_{\text{D}}$ の値をもつことが求められているが，積分は $(A \times \Delta A) \to \Delta B$ の関数を対象としており，型が合わない．詳しい関係性についての議論は今後の課題である．

8 関連研究

本研究は漸増計算との関わりが深い．漸増計算については多数の研究の歴史があるので，ここでは特に関連の深い研究について挙げる．

本研究は Cai ら漸増的ラムダ計算 [2] を基礎として

いる．Cai らは構文的なラムダ式の変換として微分（本研究で言うところの差分）を定義した．$f :: A \to B$ の差分関数の型が $(A, \Delta A) \to B$ となるところも含め，本研究では基本的にこの手法を踏襲している．差異はパラメトリシティを用いている点だが，本研究で導出される差分関数は Cai らの微分によって得られる関数と意味上一致する．ただし，Cai らは単純型付きラムダ計算を考えているのに対し，本研究ではパラメトリシティが成り立つ範囲で任意のプログラムを考えている点が異なる．

Giarrusso ら [5] は Cai らの手法を再帰関数を含むラムダ計算に拡張し，さらに効率改善のためにある種のメモ化を加えた．本研究では議論を単純化するために Giarrusso らの提案は取り入れていないが，実用上はメモ化は重要である．メモ化を本研究の定式化で素直に導入できるかの検討は今後の課題である．ただし，Giarrusso らはプログラムが A 正規形（の変種）であり全ての変数が lambda lifting されていることを仮定しており，本研究が一般的なプログラムを想定しているのとは状況が異なることは注意すべきである．

漸増的ラムダ計算の考え方は，例えば漸増的双方向変換手法の導出 [6] などに利用されている．本研究の議論がこれら実用的な問題に適用できるかの確認も重要な課題である．

漸増的ラムダ計算の顕著な特徴は，静的な手法であること，そして高階関数を扱えることである．既存の漸増計算手法の多くは動的である．特に，高階関数を扱える動的な漸増計算手法としては self-adjusting computing [1] が代表的である．静的で高階関数を扱える漸増計算手法としては，他にも部分評価に基づくもの [14] がある．本研究との関連や融合の可能性についての調査は今後の課題である．

高階関数型言語に微分を取り入れる他の試みとしては，Ehrhard と Regnier による differential lambda calculus [3] が挙げられる．彼らの体系が，微分を複雑な計算の局所的な線形近似と捉えることで，線形性を中心として構成されているのに対し，本研究では漸増計算，つまり差分化から議論を始め，線形性については全く触れなかった．そのため，両者には直接の関係はないように見える．

本研究は，差分化から始め，極限を取ることで Elliott の前向き自動微分 [4] を導出した．前向き自動微分には他の手法もある．特に Siskind と Pearlmutter [13] による手法は，小さな変化を表す記号を伝播する点で，より差分化に近い．実際，Siskind と Pearlmutter の手法と漸増的ラムダ計算を融合する試みもなされている [7]．しかし，Siskind と Pearlmutter の手法は理論的な取り扱いが難しいことが知られている [9]．本研究では差分と微分を理論的に見通しよく繋ぐことに成功している．

著者の知る限り，漸増計算や微分に比べて，和分や積分についての議論は少ないようである．数少ない例外である Morihata [10] の漸増計算手法についてはすでに述べた．他の可能性については今後調べてゆきたい．

9 まとめと今後の課題

本研究では漸増的ラムダ計算を基礎に，これを多相型のパラメトリシティの観点から再定式化することで，微分・積分へと議論を発展させた．特に，漸増的ラムダ計算の差分化が一般的なプログラムに適用できることを示したこと，同様の手法でプログラムの微分が得られること，さらにそれら議論が和分・積分へ繋がることを示したことは主要な貢献である．

本研究はプログラムの微分・積分を考える上での萌芽的なものであり，今後の課題は多数残っている．

まず，本研究では基本的に 1 引数関数の微積分を考えた．微積分学では多引数関数の微積分が非自明であることが知られている．本研究の手法は全微分へはほぼ自明に拡張できる．しかし，偏微分についての議論は今後の課題である．

プログラムに関して微分積分が適切に定義できた暁には，既存の微分積分学での成果を援用できると望ましい．特に，テイラー展開や解析接続は，プログラムの性質の解析等において有用であると期待される．これらの可能性の検討についても今後の課題である．

本研究では，プリミティブ演算が一定の性質を満たす場合に，それを用いて構成されたプログラムが同様の性質を維持する，という構造に注目し議論を行っ

た．この構造を用いることで，他にも様々な変換が可能であることが期待される．例えば，自動微分の文脈では，今回考えた前向き自動微分だけでなく，後ろ向き自動微分なども存在し，機械学習の文脈等で重要となっている．後ろ向き自動微分を同様に扱う手法は今後の課題である．他の例として，例えば凸関数であれば大域最小値をニュートン法などで簡単に求めることができるが，これをプログラムの性質として扱うことも興味深い可能性だと考えている．

謝辞　関連研究を紹介してくださった北京大学の胡振江教授に感謝する．本研究は JSPS 科研費 19K11896 および 2019 年度国立情報学研究所公募型共同研究（19FA01）の助成を受けている．

参 考 文 献

[1] Acar, U. A., Blelloch, G. E., and Harper, R.: Adaptive functional programming, *ACM Trans. Program. Lang. Syst.*, Vol. 28, No. 6(2006), pp. 990–1034.

[2] Cai, Y., Giarrusso, P. G., Rendel, T., and Ostermann, K.: A theory of changes for higher-order languages: incrementalizing λ-calculi by static differentiation, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, O'Boyle, M. F. P. and Pingali, K.(eds.), ACM, 2014, pp. 145–155.

[3] Ehrhard, T. and Regnier, L.: The differential lambda-calculus, *Theor. Comput. Sci.*, Vol. 309, No. 1-3(2003), pp. 1–41.

[4] Elliott, C. M.: Beautiful differentiation, *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, Hutton, G. and Tolmach, A. P.(eds.), ACM, 2009, pp. 191–202.

[5] Giarrusso, P. G., Régis-Gianas, Y., and Schuster, P.: Incremental \lambda -Calculus in Cache-Transfer Style - Static Memoization by Program Transformation, *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*, Caires, L.(ed.), Lecture Notes in Computer Science, Vol. 11423, Springer, 2019, pp. 553–580.

[6] Horn, R., Perera, R., and Cheney, J.: Incremental relational lenses, *PACMPL*, Vol. 2, No. ICFP(2018), pp. 74:1–74:30.

[7] Kelly, R., Pearlmutter, B. A., and Siskind, J. M.: Evolving the Incremental λ Calculus into a Model of Forward Automatic Differentiation (AD), *CoRR*, Vol. abs/1611.03429(2016). Extended abstract presented at the AD 2016 Conference, Sep 2016, Oxford UK.

[8] Koch, C., Ahmad, Y., Kennedy, O., Nikolic, M., Nötzli, A., Lupei, D., and Shaikhha, A.: DBToaster: higher-order delta processing for dynamic, frequently fresh views, *VLDB J.*, Vol. 23, No. 2(2014), pp. 253–278.

[9] Manzyuk, O.: A Simply Typed λ-Calculus of Forward Automatic Differentiation, *Electr. Notes Theor. Comput. Sci.*, Vol. 286(2012), pp. 257–272.

[10] Morihata, A.: Incremental computing with data structures, *Sci. Comput. Program.*, Vol. 164(2018), pp. 18–36.

[11] Peyton Jones, S.(ed.): *Haskell 98 Language and Libraries: The Revised Report*, Cambridge University Press, Cambridge, UK, 2003.

[12] Reynolds, J. C.: Types, Abstraction and Parametric Polymorphism, *Information Processing*, Vol. 83(1983), pp. 513–523.

[13] Siskind, J. M. and Pearlmutter, B. A.: Nesting forward-mode AD in a functional framework, *Higher-Order and Symbolic Computation*, Vol. 21, No. 4(2008), pp. 361–376.

[14] Sundaresh, R. S. and Hudak, P.: Incremental Compilation via Partial Evaluation, *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991*, Wise, D. S.(ed.), ACM Press, 1991, pp. 1–13.

[15] Wadler, P.: Theorems for Free!, *FPCA'89 Conference on Functional Programming Languages and Computer Architecture. Imperial College, London, England, 11-13 September 1989*, ACM, New York, 1989, pp. 347–359.