DBSJ Journal, Vol.9, No.3
February 2011

遅延演算を利用した再帰的ストリームデータ処理の計算完備性

Turing Completeness of Recursive Stream Data Processing with Delay Operators

今木 常之[♠]　西澤 格[♠]

Tsuneyuki IMAKI　Itaru NISHIZAWA

株自動取引，車両位置情報，携帯電話操作ログなど，膨大な情報流を継続的に解析し，リアルタイムに社会行動に結びつける技術としてストリームデータ処理が注目されている．ストリームデータ処理では宣言型のクエリ言語によりデータ処理内容を容易に定義可能である一方，株取引のように複雑な処理が必要となる応用では，クエリ言語に高い表現能力が求められる．我々は，クエリ言語の表現能力向上を目的として，再帰処理を実現するDTS法を提案した．クエリの再帰的定義は自己参照を含むため論理矛盾を引き起こす．これに対し我々は，再帰クエリに遅延演算を挿入することで，自己参照関係を解消する方法を提案している．本稿では，DTS 法で実現する言語が計算完備であることを証明し，ストリームデータ処理の適用範囲が十分広範であることを示す．

Automated trading systems, car location systems, and cell-phone systems produce huge amount of time series data. Stream data processing is expected as a new technology which analyzes the data continuously to enable real-time feedback to the social activity. Users can define their application easily using declarative languages of stream data processing systems. On the other hand, the language should equip sufficient expressive power to define complex processes such as automated trading. We proposed a method named DTS to execute recursive queries on stream data to expand the expressive power of the query language. Recursive query definition involves a self-reference which causes a paradox in the query processing. To cope with the problem, we introduced a method which inserts delay operators in recursive queries to resolve the self-reference. In this article we prove that the query language implemented by DTS is Turing complete and show that the stream data processing system with our method equips sufficient expressive power enough to be applied to a wide variety of applications.

1. はじめに

株自動取引，携帯操作，センサデータなど，社会活動で継続的に発生する情報流から有意義なイベントの発生をいち早く抽出し利活用する技術として，ストリームデータ処理（以下，ストリーム処理）が注目されている．ストリーム処理では関係データベースのクエリ言語SQLを拡張した宣言型言語によりデータ処理を定義可能であるため，アプリケーションの開発が容易である点が一つの特長となっている[1]．ストリーム処理におけるクエリ言語の多くは，関係代数に対して無限に続くデータ列から演算対象を切り出すウィンドウ演算を追加した形態を取る．ユーザはこの演算を活用することで，イベント間の発生間隔，因果関係，統計値の時間変化といった，時間の流れに基づくデータ処理を簡潔に記述できる．我々は，米Stanford大で提案された言語CQL（Continuous Query Language）[2]を採用している．CQLは関係演算の結果集合における変化分のみを抽出して正規化するストリーム化演算を備える．この演算には結果の一意性を保証しつつ意味のある変化のみを抽出する効果がある．

ストリーム処理の有望な適用先として，株価の瞬間的な変化を捉えて発注を繰り返す株自動取引が挙げられる．発注の際には，保有する株や資金の情報も重要である一方，これらは発注自体によっても変化する．このような自己参照性をCQLで扱うために，我々は，遅延演算を利用した再帰処理方法であるDelay based Time-Splitting execution（DTS）法を提案し，非再帰処理と同程度の性能で再帰処理を実現可能であることを確認した[8]．再帰処理によってストリーム処理の適用範囲が拡大する一方で，その範囲を明確に示しておくことは，実業務への適用可否を判断するうえで必須である．

本稿では，DTS法による再帰処理方法の正当性を示す．更に，DTS法で実現する再帰処理を備えたCQLが，計算可能な任意の処理を表現可能であること，即ち計算完備であることを示す．以上により，ストリーム処理の適用範囲に機能的な制限が無いことを示す．

2. 遅延演算を利用した再帰処理の正当性

本章では，提案したDTS法の再帰的なストリーム処理が，CQLの意味論に照らして健全であることを示す．まず，DTS法の概要を2.1節に示す（詳細は[8]を参照）．また，DTS法においてクエリ解析時に決定する実行順序に従ってオペレータを処理することで，正しい結果が得られることを2.2節に示す．さらに，ユーザが定義したクエリに実行不可能な再帰ループが含まれる場合には，クエリ解析時に必ず検出可能であることを2.3節に示す．

2.1 提案手法の概要

再帰的なクエリ定義とは，あるクエリが，FROM句に指定する入力としてそのクエリ自身を，あるいはそのクエリを入力とする他のクエリを取るような形で定義されている場合を指す．このように定義されたクエリを，オペレータをノード，オペレータ間の入出力キューを有向辺とするクエリグラフに変換した場合，強連結成分（辺を辿ることで互いに到達可能なノードの集合）を含む有向グラフとなる．

CQLの意味論は，時間軸で切ったデータ集合に対して関係代数を適用するモデルであり[2][7]，あるオペレータの入力と出力がループする場合，そのオペレータの処理結果は，生成されると同時に同オペレータに入力されることになる．このような意味論においては，強連結成分のノードとして非単調なオペレータが含まれると実行状態が無限に振動するという問題があった．なお，ここでの単調性とは，そのオペレータの入力データ集合に新しい要素が加わった場合に結果のデータ集合からいかなる要素も消滅しないことを意味しており，非単調なオペレータとはこの性質を充たさないものを指す．MAX，SUMなどの集約演算，差集合演算，個数ウィ

♠ 正会員　株式会社日立製作所中央研究所
{tsuneyuki.imaki.nn, itaru.nishizawa.cw}@hitachi.com

ンドウ，DStreamなどは非単調である．

[8]では個数ウィンドウを用いた株自動取引の再帰クエリを例に説明している．株取引においては，株価情報のみならず，資金や保有株などの情報も考慮する必要がある．一方で，これらの情報は発注行為自体によっても変化する．株価情報が外部入力であるのに対し，資産などの情報は処理に伴って随時変化する内部状態と見做すことができる．個数ウィンドウはこういった状態を保持するオペレータとして適当であるものの，先述した実行状態の振動が発生するため，状態保持の目的では利用不可能であった．

以上の課題に対し，[8]ではストリームタプルのタイムスタンプを未来にずらす遅延演算を再帰ループに挿入し，非単調オペレータが出力するタプルの時刻と，同タプルがクエリグラフを巡って再び同オペレータの入力に戻る時刻がずれるようにすることで，振動の発生を回避するDTS法を提案した．遅延演算の挿入位置は，自動ではなくクエリの意味を考慮してクエリ定義者が決定する方針とした．DTS法は，適切に遅延演算が挿入された再帰クエリを，CQLの意味論を保存しつつ実行することを目的としている．

DTS法では，クエリ解析時に遅延演算の挿入位置に基づいてオペレータ間の実行順序を決定し，クエリ実行時にはその順序に従ってクエリグラフ上の実行可能なオペレータを処理する．また，個別のタイムスタンプ（以下，実行時刻）毎に，同じ時刻の付いたグラフ上のタプルを全て処理し尽くすように実行する．このように，実行時刻毎に処理を完結すること，および遅延演算でクエリグラフを時間的に分割することを意味して，DTS法の名前を定めた．

ある実行時刻の処理において，当該時刻を付与されたタプルが遅延演算に入力されると，同遅延演算は当該時刻より後の時刻を付与して同タプルを出力するので，当該時刻の実行は同遅延演算で停止することになる．これに基づき，遅延演算が起終点となるようにオペレータの実行順序を決定する．

2.2 オペレータ実行順序の妥当性

DTS法の実行順序決定アルゴリズムを図1に示す．

```
1: numbering(G(V, E, f), D, v_s, s)
2:   // G(V, E, f): 有向グラフ(V はノードvの集合，E は辺eの集合,
3:   //              f: E→V×V は辺から始点×終点ノードへの写像)
4:   // D: 遅延演算ノードの集合，  v_s: 開始ノード，  s: 開始番号
5:   E_d := E − {e | f(e) = (v_x, v_s)} // 開始ノードに向かう辺を除去
6:   CSet := divide(G(V, E_d, f))       // 強連結成分Cの集合に分解
7:   CList := sort(CSet, E_d, f, v_s)   // 辺の方向に従い成分を整列
8:                                      // CList = {C_1, ..... , C_m}
9:   for i := 1 to m do
10:      D_i := C_i∩D
11:      if  |C_i| =1 or D_i = φ   then   set_number(C_i, s)
12:      else
13:          E_i := {e | f(e) = (v_a, v_b), v_a∈C_i, v_b∈C_i}
14:          numbering(G(C_i, E_i, f), D_i,  ∀v_i∈D_i, s)
15:      end if
16:      s := s+|C_i|
17:  end do
```

図 1　実行順序決定アルゴリズム
Fig.1　The algorithm to decide execution order in DTS

クエリ定義を解析して得られるクエリグラフに対し，ダミーの開始ノードv_0を追加し，同グラフのリーフノードとなる全てのストリーム入力オペレータ（Scanオペレータ）に向けてノードv_0から辺を張る．こうして得られた有向グラフを$G(V, E, f)$とし，クエリグラフに含まれる全遅延演算ノードをDとして，引数の組$(G(V, E, f), D, v_0, 0)$でnumberingを呼ぶことにより，オペレータの実行順序を決定することができる．

このアルゴリズムの動作の直感的な説明は，次のようになる．遅延演算でクエリグラフを切断することによってループを開放し，その結果として得られるグラフを強連結成分分解し，各成分（特別に一オペレータのみで構成されるものも成分と呼ぶ）をノードとして，当該遅延演算（起点）から同じ遅延演算（終点）に向かって，昇順に実行順序を付与する．開放したグラフにおいて更に複数ノードを要素とする部分的な強連結成分が存在する場合は，再帰的に分解を繰り返す．

6行目のdivideは有向グラフを強連結成分に分解する関数であり，その結果として得られる強連結成分の集合*CSet*を返戻する．各要素$C \in CSet$は，各成分を構成するノードの集合である．有向グラフの強連結成分分解法としてはTarjanによる線形時間のアルゴリズム[5]が知られている．また，7行目のsortは，二つの強連結成分を跨った辺の方向に沿って定まる半順序関係に従って，全成分を整列する関数を想定している．即ち，以下の条件1を充たす*CList*を返戻する．

条件1

$CList = \{C_1, \ldots\ldots, C_m\}$において，
$\forall v_a \in C_i,\ \forall v_b \in C_j (0 \leq i < j \leq m)$について，$f(e) = (v_b,\ v_a)$ であるような辺eはE_dに含まれない．

なお，v_s が開始点であるため $C_1 = \{v_s\}$ となる．半順序に基づく整列であるため，直接にも間接にも順序関係の存在しない成分同士の整列順については任意性がある．但し，この任意性はDTS法の健全性には影響しない（理由は後述の定理1による）．sortのアルゴリズムは紙面の都合で割愛する．

11行目のset_numberは，第一引数の強連結成分C_i に含まれる各ノードに対して，第二引数の開始番号sから始まる一意の番号を付与する関数である．ここで付与された番号が，クエリ実行時におけるオペレータの実行順序となる．C_i のサイズが1である場合は，その構成要素である唯一のオペレータに対して実行順序sを付与する．一方，サイズが1より大きい場合は，$D_i = \phi$ のとき，即ちC_i が遅延演算ノードを含まないときに限ってset_numberが処理される．このケースについては2.3節で述べる．

13,14行目は，部分的な強連結成分への付番のために再帰的に本アルゴリズムを呼び出す．13行目はC_i に含まれるノードで構成される部分グラフの辺を抽出する．また，14行目の再帰呼出し時の引数として，遅延演算の中から選択する開始ノードv_iには任意性がある．但し，この任意性はDTS法の健全性には影響しない（理由は後述の定理1，および2.3節に示す定理2による）．

各成分C_iに付与する実行順序の開始番号sは，ノードに付与する実行順序が重ならないよう，16行目で成分のサイズ$|C_i|$だけ増加させる．従って，以下の条件2を充たす．

条件2

$CList = \{C_1, \ldots\ldots, C_m\}$において，
$\forall v_a \in C_i,\ \forall v_b \in C_j (0 \leq i < j \leq m)$について，$n(v_a) < n(v_b)$.
（但し，$n(v)$はノードv に付与された実行順序とする）

以上のアルゴリズムによって決定される実行順序が妥当であることを示すために，以下の定理1を証明する．

定理1

クエリグラフの辺のうち，始点ノードの実行順序が終点ノードの実行順序よりも大となるのは，終点ノードが遅延演算であるもののみである．

証明

条件1および条件2が充たされるため，$n(v_a) < n(v_b)$であるノード$\forall v_a$，$\forall v_b$について，$f(e) = (v_b, v_a)$であるような辺eはE_dに含まれない．一方，E_dに含まれない可能性があるのはアルゴリズム5行目で除外される辺のみであり，その終点は開始ノードv_sである．14行目において，開始ノードv_sは遅延演算から選ばれる． □

図2に示した例は，同じクエリグラフ（[8]より引用）に対して，開始ノードの選択が異なる二つのケースにおいて決定された実行順序である．太枠のノードは遅延演算を表す．ケース(1)における順序の決定過程は[8]に示した通りである．ケース(2)も同様に，まずeで切断し$CList$ = { {e}, {c}, {m}, {a, b, f, g, h, i, j, k, l, n}, {d} }を得て，その4番目の要素をgで切断し$CList$ = { {g}, {a, b, h, i, j, k, l, n}, {f} }，その2番目の要素をaで切断し$CList$ = { {a}, {b}, {h}, {i, j, k}, {l}, {n} }を得る．始点ノードの実行順序の方が終点ノードよりも大となっている辺を太線で示した．ケース(1)と(2)は共に，そのような辺の終点ノードが遅延演算となっており，定理1が成立している．

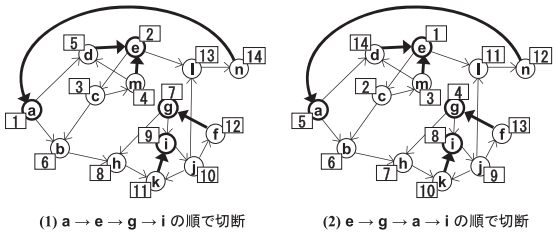


図 2　実行順序の任意性
Fig.2　Facultativity on Operator Execution Orders

定理1によって，2.1節に示した実行時刻毎のクエリグラフの処理が，遅延演算を除いてオペレータ間の入出力関係通りに進行することを示した．即ち，各オペレータの実行時には，当該実行時刻において処理すべき入力データが全て揃うことが保証される．一方，[2][7]に示されたCQLの意味論に従えば，ある時刻における処理の健全性は，時間軸上の当該時刻で入力／出力リレーションを切った交点の集合の同一性に基づく．オペレータ実行時に入力データが揃うことからこの同一性が保存されるので，DTS法の処理方法は健全である．

なお，当該実行時刻において遅延演算に向けて出力されたデータの処理は，2.1節に示した通り一旦停止する（遅延演算に入力されるのみで，出力はされない）ため，遅延演算は当該実行時刻の処理の最後に纏めて実行して構わない．

2.3 実行不可能な再帰ループの検出

2.1節に示したように，ユーザが定義したクエリにおいて，非単調なオペレータを含む再帰ループが存在する場合には，発振するため実行不可能である．また，ストリーム化演算を含む場合にも実行不可能である．理由は，[2][7]に示されたストリーム化演算の定義に従うと，ある時刻におけるストリーム化演算の出力は，当該時刻の入力データが全て揃わないと確定しないためである．この動作によって，入力に含まれるゴースト（生存期間がゼロである，意味の無いリレーション）を除去することが可能となっているが，ストリーム化演算が再帰ループに含まれる場合は入出力の因果関係が短絡するためハングアップすることになる[1]．また，ストリーム化演算を入力とするウィンドウ演算も，同様に再帰ループに含めることができないことになる．

但し，これまで示したように，上記のような再帰ループが遅延演算も同時に含むのであれば実行可能となる．従って，処理の健全性を保証するためには，遅延演算を含まず，かつ実行不可能なオペレータを含む再帰ループを検出できる必要がある．図1のアルゴリズムによってこの検出が可能であることを示すために，以下の定理2を証明する．

定理2

アルゴリズム停止時点で分解されずに残る強連結成分の集合は，アルゴリズムの任意性に関わらず同一である．

補題

アルゴリズムに入力されるクエリグラフが，遅延演算を含まない部分的な強連結成分C_xを内包するならば，アルゴリズムの任意性に関わらず，アルゴリズム停止時点までC_xは分解されない．

補題の証明

アルゴリズムの過程で削除する辺は，5行目の遅延演算を終点とする辺eのみである．一方，C_xは遅延演算を含まないため，辺eはC_xのノード間を結ぶ辺ではない．即ち，辺eを削除してもC_xのノード間の接続関係は保存される．従って，divideはC_xを分割しない． □

定理の証明

アルゴリズム停止時点で分解されずに残る各強連結成分は，11行目の判定において遅延演算を含まないために真となってset_numberに渡されるものであり，それ以上分解されることはない．即ち，強連結成分として最小のノード集合である．また補題より，このような強連結成分はアルゴリズムの任意性に関わらず全て分解されずに残る．最小サイズの強連結成分が全て残ることになるため，結果として得られる成分の集合は，アルゴリズムの任意性に関わらず同一となる． □

図2に示した例において，もしノードiが遅延演算ではないならば，ケース(1)と(2)では共に{i, j, k}が分解されずに残る．

定理2によって，遅延演算を含まない再帰ループを確実に検出可能であることを示した．これらのループが実行不可能な演算を含むか否かは容易に判定可能である．なお，実行不可能な演算を含まない場合の実行方法の議論は割愛する．

3. 遅延演算を利用した再帰処理の計算完備性

本章では，DTS法の計算能力の評価として，計算完備であることを示す．3.1節では，再帰クエリによって帰納的関数が実現可能であることを示す．3.2節では，帰納的関数の実現例として，再帰クエリによるAckermann関数の実現方法を示す．

3.1 再帰クエリによる帰納的関数の実現

ある計算機言語が帰納的関数を計算可能であるならば，その言語は計算完備であることが知られている（[6]など）．本稿では，DTS 法が計算完備であることの証明として，再帰クエリによる帰納的関数の構成方法を示す．

[1] IStream は単調なオペレータなので，ゴーストの取り消しを許すならばハングアップは回避される．但し，取り消しを許可すると(1)ウィンドウ演算の処理が複雑になる，(2)結果を利用するアプリケーション側で取り消しを認識する必要がある，といった悪影響があるため許可しない方針とした．

まず，帰納的関数の定義，およびその基となる原始帰納的関数の定義を以下に示す（$\boldsymbol{N}$は自然数）．

［原始帰納的関数］

(1) 零関数 zero: $\boldsymbol{N}^0 \to \boldsymbol{N}$　　但し zero() = 0,
後者関数 suc: $\boldsymbol{N} \to \boldsymbol{N}$　　但し $suc(x) = x+1$,
射影関数 p^n_i: $\boldsymbol{N}^n \to \boldsymbol{N}$　　但し $p^n_i(x_1, x_2, \cdots, x_n) = x_i$, $1 \leq i \leq n$
以上の初期関数は原始帰納的関数である．

(2) g: $\boldsymbol{N}^m \to \boldsymbol{N}$ と g_j: $\boldsymbol{N}^n \to \boldsymbol{N}$ (j=1, 2, ⋯, m)が原始帰納的関数のとき，次の関数 f: $\boldsymbol{N}^n \to \boldsymbol{N}$ は原始帰納的関数である．
$f(\boldsymbol{x}) = g(g_1(\boldsymbol{x}), g_2(\boldsymbol{x}), \cdots, g_m(\boldsymbol{x}))$　　($\boldsymbol{x}$ は $x_1, x_2, \cdots, x_n$)

(3) g: $\boldsymbol{N}^n \to \boldsymbol{N}$ と h: $\boldsymbol{N}^{n+2} \to \boldsymbol{N}$　が原始帰納的関数のとき，次の関数 f: $\boldsymbol{N}^{n+1} \to \boldsymbol{N}$ は原始帰納的関数である．
$f(\boldsymbol{x}, 0) = g(\boldsymbol{x})$
$f(\boldsymbol{x}, y+1) = h(\boldsymbol{x}, y, f(\boldsymbol{x}, y))$

(2)は関数の合成，(3)は原始帰納法による関数定義である．

［帰納的関数］

(1) 原始帰納的関数は帰納的関数である．

(2) 帰納的関数の合成関数は帰納的関数である．

(3) 帰納的関数から原始帰納法によって得られる関数は帰納的関数である．

(4) p: $\boldsymbol{N}^{n+1} \to \{0, 1\}$が原始帰納的関数のとき，次の関数 μ_y: $\boldsymbol{N}^n \to \boldsymbol{N}$ は帰納的関数である．

$$\mu_y(\boldsymbol{x}) = \begin{cases} p(\boldsymbol{x}, y) = 0 \text{ となる自然数 } y \text{ があればその最小値} \\ p(\boldsymbol{x}, y) = 0 \text{ となる自然数 } y \text{ がないとき，未定義} \end{cases}$$

(4)の関数を最小解関数と呼ぶ．最小解関数の定義域は $\boldsymbol{N}^n$ の部分集合となるため，帰納的関数は部分関数となる[2]．

最初に，上記で定義される帰納的関数の各構成要素がCQLの再帰クエリによって定義可能であることを示す．以降では，各構成要素の入出力は全て（リレーションではなく）ストリームとする．また，関数 f に入力されるストリーム，あるいはその入力ストリームを生成するクエリの名称を“f_in”とし，結果を出力するクエリの名称を“f”とする．関数 f の出力がそのまま関数 g の入力となる場合には，関数 g の定義における From 句にて“f AS g_in”と記載する．結果出力クエリのスキーマは一つの“v”という名称のカラムのみを含む．一方，関数への入力のスキーマは，関数の引数のみを丁度含むように定義する．なお，カラム“x1, xn”の並びは上記定義における引数の並び $\boldsymbol{x}$ を意図している．

```
REGISTER STREAM zero_in(dummy INTEGER);
REGISTER QUERY zero
 ISTREAM(SELECT 0 AS v FROM zero_in[NOW]);

REGISTER STREAM suc_in(x INTEGER);
REGISTER QUERY suc
 ISTREAM(SELECT suc_in.x+1 AS v FROM suc_in[NOW]);

REGISTER STREAM p_in(x1 INTEGER, x2 INTEGER, ..., xn INTEGER);
REGISTER QUERY pi
 ISTREAM(SELECT p_in.xi AS v FROM p_in[NOW]);
```

図 3 初期関数の CQL 表現
Fig.3　Definitions of Initial Functions in CQL

CQL による零関数，後者関数，および射影関数の表現を図3に示す．これらは単純に一入力に対する射影演算として，SELECT 句と FROM 句のみで定義される．零関数は本来引数を取らないが，CQL においてはカラム値を持たないタプルを扱うことができないため，入力ストリームにはダミーのカラムを一つ設けている．

次に，CQL による関数合成の表現を図 4 に示す．図の左側がクエリ定義であり，右側がクエリ間の接続関係を表すグラフである．また，クエリ定義中の“<<”と“>>”で挟まれた部分には各副関数の定義が入ることを想定する．グラフにおいては，名称を斜体で示した六角形の箇所がそれら副関数に相当する．以降の図 5 と図 6 においても同様である．

関数 g_j: $\boldsymbol{N}^n \to \boldsymbol{N}$　(j=1, 2, ⋯, m)の各々について，g***j***_in, g***j***，および g_in_***j*** の 3 つのクエリを定義する．但し，g1_in の代わりに f_in とし，g_in_m の代わりに g_in とする．g_in_***j*** は，g***j*** によって計算された関数 g_j の結果を，最終的に関数 g に入力するために一つずつ追加していくクエリである．従って，g_in_***j*** の結果は j 個のカラム(y1, y2, ⋯, y***j***)を含むことになる．

```
REGISTER STREAM f_in(x1 INTEGER, xn INTEGER);

<< REGISTER QUERY g1 ...
                FROM f_in AS g1_in ... >>

REGISTER QUERY g_in_1
 ISTREAM(
  SELECT g1.v AS y1
    FROM g1[NOW]);

REGISTER QUERY g2_in
 ISTREAM(
  SELECT f_in.x1, f_in.xn
    FROM g1[NOW], f_in[ROWS 1]);

<< REGISTER QUERY g2 ... FROM g2_in ... >>

REGISTER QUERY g_in_2
 ISTREAM(
  SELECT g_in_1.y1, g2.v AS y2
    FROM g2[NOW], g_in_1[ROWS 1]);

.........

REGISTER QUERY gm_in
 ISTREAM(
  SELECT f_in.x1, f_in.xn
    FROM gm-1[NOW], f_in[ROWS 1]);

<< REGISTER QUERY gm ... FROM gm_in ... >>

REGISTER QUERY g_in
 ISTREAM(
  SELECT g_in_2.y1, g_in_2.y2, ..., gm.v AS ym
    FROM gm[NOW], g_in_m-1 [ROWS 1]);

<< REGISTER QUERY g ... FROM g_in ... >>

g AS f
```

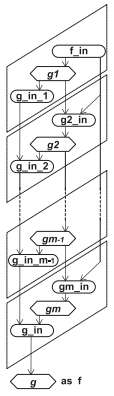


図 4 関数合成の CQL 表現
Fig.4　A Definition of Composition in CQL

次に，CQL による原始帰納法の表現を図 5 に示す．この定義は再帰クエリとなる．再帰ループの起点となるクエリ w は ($\boldsymbol{x}$, y, v = $f(\boldsymbol{x}, y)$)の三項組を出力する．この値はループを一周する過程でクエリ h に入力し，そのクエリの処理結果として $h(\boldsymbol{x}, y, f(\boldsymbol{x}, y))$の値を得る．これは $f(\boldsymbol{x}, y+1)$の値であるので，クエリ f_y によってインクリメントした y の値と組み合わせてクエリ w に戻す．即ち，y を状態変数としてインクリメントしつつ，求めるべき y の値 yr に達するまでループを回すという動作である．y = yr が充たされた際に，クエリ f が結果を出力する．クエリ f_0 は y=0 における値，即ち $f(\boldsymbol{x}, 0) = g(\boldsymbol{x})$を算出し，その値が再帰ループの初期値となる．

ここで，クエリ f_y において w の生存期間を[ROWS 1]の個数ウィンドウで定めているのは，最新の y の値のみを状態として保持するためである．これによって，クエリ h の結果は最新の y 一つと結合される[3]．2.1 節で述べた個数ウィンドウによる振動を避けるために，再帰ループには遅延演算が挿入される必要があり，ここではクエリ f_y の出力に位置している（グラフでは太枠丸の位置）．同クエリ定義の末尾に

[2]本稿では部分帰納的関数の意味で帰納的関数と呼ぶ．

[3] 副関数 h の計算において，カラム x, y, yr の値を計算過程の全クエリで引継ぎ，クエリ h の結果として v の他にもこれらのカラムの値を再び獲得できるようにクエリを構成すれば，クエリ f_y における w との結合は不要となり[ROWS 1]のウィンドウも不要になる．従って IStream や遅延演算も不要になる．同様の考え方はクエリ f_0，および図 4 と図 6 に示したクエリにも当てはまる．但し，そのような構成方法では計算過程の全ての中間状態が同時に生存して残ることになるため，空間コストが問題になる．

ある“<”から“>”までが遅延演算の指定であり，delay-sizeは遅延期間である．delay-sizeには，時間ウィンドウの一種である[NOW]の期間（実時間の最小単位よりも小さい仮想的な時間[8]）の2倍以上，あるいは実時間を指定する．delay-sizeを[NOW]の期間丁度に指定すると，関数hの計算結果が連続するyで同じ値を取る場合にIStream演算の作用によってクエリhの結果が出力されなくなるため，これを回避する目的でdelay-sizeを[NOW]より長い期間にとる必要がある．

次に，CQLによる最小解関数の表現を図6に示す．この定義は再帰クエリであり，クエリの基本的な構造は図5と同様である．即ち，wを再帰ループの起点として，yの値をインクリメントしつつ関数$p(x, y)$の値が0となるまでループを回すという動作である．ここで，$p(x, y) = 0$となるyが存在しない場合は無限ループとなるため，結果が未定義となる．

```
REGISTER STREAM
      f_in(x1 INTEGER, xn INTEGER, yr INTEGER);

REGISTER QUERY g_in
 ISTREAM(
  SELECT f_in.x1, f_in.xn
    FROM f_in[NOW]);

<< REGISTER QUERY g ... FROM g_in ... >>

REGISTER QUERY f_0
  SELECT g.v, f_in.x1, f_in.xn, 0 AS y, f_in.yr
    FROM g[NOW], f_in[ROWS 1];

REGISTER QUERY w
 ISTREAM(
  SELECT * FROM f_0
 UNION ALL
  SELECT * FROM f_y[NOW]);

REGISTER QUERY h_in
 ISTREAM(
  SELECT w.x1, w.xn, w.y, w.v
    FROM w[NOW]
   WHERE w.y <> w.yr);

<< REGISTER QUERY h ... FROM h_in ... >>

REGISTER QUERY f_y
 ISTREAM(
  SELECT h.v, w.x1, w.xn, w.y+1 AS y, w.yr
    FROM h[NOW], w[ROWS 1])<delay-size>;

REGISTER QUERY f
 ISTREAM(
  SELECT w.v
    FROM w[NOW]
   WHERE w.y = w.yr);
```

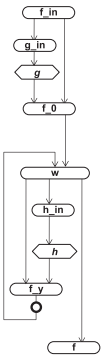


図5　原始帰納法のCQL表現
Fig.5　A Definition of Primitive Recursion in CQL

```
REGISTER STREAM u_in(x1 INTEGER, xn INTEGER);

REGISTER QUERY w
 ISTREAM(
  SELECT u_in.x1, u_in.xn, 0 AS y
    FROM u_in[NOW]
 UNION ALL
  SELECT * FROM u_y[NOW]);

<< REGISTER QUERY p ... FROM w AS p_in ... >>

REGISTER QUERY u_y
 ISTREAM(
  SELECT w.x1, w.xn, w.y+1 AS y
    FROM p[NOW], w[ROWS 1]
   WHERE p.v <> 0)<delay-size>;

REGISTER QUERY u
 ISTREAM(
  SELECT w.y AS v
    FROM p[NOW], w[ROWS 1]
   WHERE p.v = 0);
```

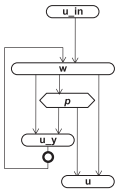


図6　最小解関数のCQL表現
Fig.6　A Definition of Minimization in CQL

以上で，帰納的関数の全ての構成要素をCQLで実現可能であることを示した．この中で，“<<”と“>>”で挟まれた部分の各副関数自体が，これら構成要素を再帰的に組み合わせることで定義可能である．その理由は，各構成要素における副関数との入出力を，リレーションではなくストリームとして，即ち期間を持たない時間軸上の点列として定義しているためである．そのことによって，任意の構成要素同士を接続可能であることが保証される．

以上より，DTS法で実現する再帰処理を備えたCQLが帰納的関数を計算可能であり，従って計算完備であることが示された．なお，自然数は任意の大きさの値が表現できることを仮定した．この仮定は，INTEGER型の替わりにDECIMAL型を利用することで実現可能である．

3.2 帰納的関数の実現例：Ackermann関数の計算

DTS法が帰納的関数を実現することの例示として，原始帰納的関数ではない帰納的関数として知られているAckermann関数（[6]など）が計算可能であることを示す．Ackermann関数の定義は次の通りである．

(0)　$f(0, y) = y+1$
(1)　$f(x, 0) = f(x-1, 1)$
(2)　$f(x, y) = f(x-1, f(x, y-1))$

前節の結果に基づけば，これを計算するプログラムをC，Javaなどの手続き型言語で定義し，その処理系をエミュレートすることでも実現可能であるが，本稿ではよりCQLでの定義として適切な方法を採用する．クエリを図7に示す．

```
REGISTER STREAM
   input(x INTEGER, y INTEGER);

REGISTER QUERY f0
 ISTREAM(
  SELECT f.y+1 AS v, f.x, f.y
    FROM f
   WHERE f.x = 0);

REGISTER QUERY f1n
 ISTREAM(
  SELECT f.x-1 AS x, 1 AS y
    FROM f
   WHERE f.y = 0);

REGISTER QUERY f1
 ISTREAM(
  SELECT r.v, f.x, f.y
    FROM f, r
   WHERE f.y = 0
     AND f.x-1 = r.x AND r.y = 1);

REGISTER QUERY sn
 ISTREAM(
  SELECT f.x, f.y
    FROM f
   WHERE f.x > 0 AND f.y > 0);

REGISTER QUERY f2n
 ISTREAM(
  SELECT f.x, f.y-1 AS y
    FROM f
   WHERE f.x > 0 AND f.y > 0);

REGISTER QUERY sf2n
 ISTREAM(
  SELECT s.x-1 AS x, r.v AS y
    FROM s, r
   WHERE s.x = r.x AND s.y-1 = r.y);

REGISTER QUERY f2
 ISTREAM(
  SELECT rs.v, f.x, f.y
    FROM f, r AS ri, r AS rs
   WHERE f.x > 0 AND f.y > 0
     AND f.x = ri.x AND f.y-1 = ri.y
     AND f.x-1 = rs.x AND ri.v = rs.y);
```

```
REGISTER QUERY r_u
 ISTREAM(
  SELECT * FROM f0[NOW]
 UNION ALL
  SELECT * FROM f1[NOW]
 UNION ALL
  SELECT * FROM f2[NOW])<NOW>;

REGISTER QUERY r
  SELECT *
    FROM r_u[PARTITION BY r_u.x, r_u.y
                                  ROWS 1];

REGISTER QUERY f_u
 ISTREAM(
  SELECT * FROM input[NOW]
 UNION ALL
  SELECT * FROM f1n[NOW]
 UNION ALL
  SELECT * FROM f2n[NOW]
 UNION ALL
  SELECT * FROM sf2n[NOW])<NOW>;

REGISTER QUERY f
  SELECT *
    FROM f_u[PARTITION BY f_u.x, f_u.y
                                  ROWS 1];

REGISTER QUERY s
  SELECT *
    FROM sn[PARTITION BY sn.x, sn.y
                                 ROWS 1];

REGISTER QUERY output
 ISTREAM(
  SELECT r_u.v
    FROM r_u[NOW], input[ROWS 1]
   WHERE r_u.x = input.x
     AND r_u.y = input.y);
```

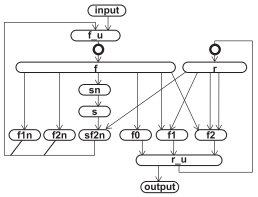


図7　Ackermann関数のクエリ
Fig.7　A Query of Ackermann Function

クエリの意図は次の通りである．求める引数(x, y)の組から関数fの計算を開始し，上記の関数定義(0)～(2)に基づいて左辺の引数の組(x, y)を右辺の組に還元して，計算が必要な組として集合Fに追加していく．一方で，ある引数組について計算結果vが得られた際には，引数と併せて(v, x, y)の組として集合Rに追加する．ここで，関数定義(0)～(2)に基づいて，右辺の値が求まれば左辺の値が連鎖的に求まる．例えば$f(3, 0)$の計算ではFに$(3, 0)$を最初に追加し，式(1)に基づいて$(2, 1)$に還元しFに追加する．計算が進んで$f(2, 1)$の値(5である)が求まれば，即ちRに$(5, 2, 1)$が追加されれば，Fにある$(3, 0)$が解決するのでRに$(5, 3, 0)$を追加する．

一方，式(2)に該当する引数組$(x>0, y>0)$がFに加わった際には，別途用意する集合Sにもそれを保存すると同時に，Fに$(x, y-1)$を加える．後に$f(x, y-1)$の値が解決された際に，その値をvとして$(x-1, v)$の組を改めてFに追加する．例えばFに$(3, 1)$が追加された際にはこれをSにも追加し，Fに$(3, 0)$を追加する．Rに$(5, 3, 0)$が追加された際に改めて$(2, 5)$をFに追加する．

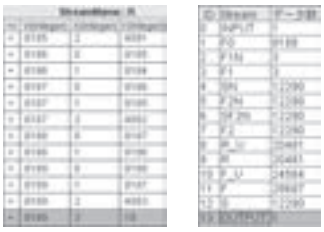
図7では，クエリr, f, sがそれぞれ上記の集合R, F, Sを表す．また，クエリf1n, f2n, sf2nは式(1),(2)の左辺から右辺への還元を，クエリf0, f1, f2は求まった式(0),(1),(2)の右辺から

左辺の決定をそれぞれ表し，前者はFに，後者はRにそれぞれクエリf_uとr_uで纏めて追加する．

クエリf1, f2, sf2nでは，集合F（またはS）とRの結合によって，集合Rへの計算結果の追加による，値の解決の連鎖を表現している．ここで，クエリr, f, sにおいて[PARTITION by x, y ROWS 1]のウィンドウで生存期間を規定しているのは，各引数組(x, y)について保持する要素を一つに限定するためである．これによって，一度値が求まった引数組の再計算が回避される，即ちキャッシュ効果が得られる[4]．

図8は$f(3, 10)$の計算結果を示す．図の左半分はクエリrの実行結果であり，最下行に$f(3, 10)$の値が求まっている．図の右半分は計算の過程で各クエリが出力したタプルの数である．クエリrの出力数20,481は，計算されたユニークな引数組の個数に相当する．また，クエリf_uとr_uの出力数の差4,103は，一度計算した引数組の結果が再度必要になった回数，即ちキャッシュヒット回数に相当する．これらの数字は，手続き型言語で再帰的な関数定義とキャッシュの利用によってAckermann関数を計算した場合と一致する．なお，キャッシュを利用しなければ44,698,325回の関数呼出しが発生する。

以上，DTS法の計算完備性を示す例として，Ackermann関数が計算可能である事を実証した．



$x = 3,\ y = 10$

図 8　Ackermann 関数の処理結果
Fig.8　Execution Results of Ackermann Function

4. 関連研究

SQL をデータマイニングやストリームデータ処理に適用するための拡張案として，[3]では SQL の Insert 文や Delete 文などのデータ操作命令で構成するユーザ定義の集約演算 UDA(User-Defined Aggregates)を導入する方針を提案している．また，UDA によるチューリングマシンの模倣が可能であることに基づき，計算完備であることを示している．但し，本方式ではテーブルに保存するテープやヘッドの状態を排他的に変更するため，CQL の意味論における個数ウィンドウと同様の動作となり，2.1 節で示した実行状態の振動に繋がる．従って，CQL の再帰クエリを実現する目的では利用不可である．これに対し，本稿では遅延演算によって振動を回避することで，CQL の意味論を保存しつつ再帰クエリを実現する．また，CQL はデータ照会命令（SELECT 文）のみで構成されるため宣言的であり，クエリ定義の容易性が高いと考える．

ストリームデータの再帰処理に関する研究として[4]が関連する．再帰処理中の二項演算で punctuation の伝搬が永久に block するという課題に対し，閉ループ内に特殊なオペレータ（Flying Fixed-Point）を設け，投機的な punctuation（以下 sp）をループ内で発生させる方式を提案している．但し，ループ内には同時に一つの sp しか流さないことを前提とするため，ループ中の分岐や多重ループを含むクエリは対象外となり，3.1，3.2 節に示したクエリの構成方法は適用不可となる．また，ウィンドウ演算の利用も不可であるため，チューリングマシンの模倣に必要となる状態変数が表現不可である．従って，計算可能なクエリの記述範囲に制限があると考える．これに対し，本稿の方法は計算完備であるため，広範な応用に適用可能である．

5. まとめと今後の課題

ストリームデータに対する再帰処理の実現方法として提案した DTS 法の正当性を示した．また，DTS 法で実現される，再帰を備えた CQL が帰納的関数を計算可能であり，計算完備であることを示した．以上により，DTS 法の適用範囲に機能的な制限が無いことを示した．一方で，再帰クエリの利用は初見では敷居が高いため，定型的な活用方法の確立が必要である．クエリのテンプレート化，CQL 記法の拡張など，利便性の向上が今後の課題となる．また，DTS 法の実行性能に関してキャッシュミス率の増加によるスループット劣化を確認している[8]．性能改善策として，従来方式であるハートビート法とのハイブリッド方式，先読みキャッシュの利用なども検討する．

【文献】

[1] Stonebraker, M., Çetintemel, U. and Zdonik, S.: “The 8 requirements of real-time stream processing”, SIGMOD Rec., Vol.34, No.4, pp.42-47 (2005).
[2] Arasu, A., Babu, S. and Widom, J.: “The CQL Continuous Query Language: Semantic Foundations and Query Execution”, Technical report, Stanford (2003).
[3] Law, Y., Wang, H. and Zaniolo, C.: “Query Language and Data Models for Database Sequences and Data Streams” , In VLDB, pp.492-503 (2004).
[4] Chandramouli, B., Goldstein, J. and Maier, D.: “On-the-fly Progress Detection in Iterative Stream Queries”, In VLDB, pp.241-252 (2009).
[5] Tarjan, R.: “Depth-First Search and Linear Graph Algorithms”, SIAM J. Comput, pp.146-160 (1972).
[6] 高橋正子: “計算論 - 計算可能性とラムダ計算 - ”, 近代科学社(1991).
[7] 今木常之, 西澤格: “ストリームデータ処理におけるデータ生存期間管理方式” , DBSJ Letters Vol.5, No.2, pp.65-68 (2006).
[8] 今木常之, 樫山俊彦, 西澤格: “遅延演算を利用したストリームデータの再帰処理方法”, DBSJ Journal Vol.8, No.4, pp.7-12 (2010).

今木 常之　Tsuneyuki IMAKI
(株)日立製作所中央研究所主任研究員．1996 東京大学大学院工学系研究科修士課程修了．情報処理学会会員．

西澤 格　Itaru NISHIZAWA
(株)日立製作所中央研究所主任研究員．1996 東京大学大学院工学系研究科博士課程修了，工学博士．2002-2003 米スタンフォード大客員研究員．ACM，情報処理学会各会員．

4 Partition ウィンドウの替わりに DISTINCT を利用することでも同様の効果を得られる．その場合は IStream や遅延演算も不要となる．ここでは計算過程をキャッシュとして残すのが目的であるため，空間コストは問題ではない．