Incremental Computing with Abstract Data Structure

A. Morihata (FLOPS 2016)

2017 プログラミング代数特論
セミナー

紹介論文

Incremental Computing with Abstract Data Structures

Akimasa Morihata(✉)

University of Tokyo, Tokyo, Japan
morihata@graco.c.u-tokyo.ac.jp

Abstract. Incremental computing is a method of keeping consistency between an input and an output. If only a small portion of the input is modified, it is natural to expect that the corresponding output can be obtained more efficiently than full re-computation. However, for abstract data structures such as self-balancing binary search trees, even the most primitive modifications may lead to drastic change of the underlying structure. In this paper, we develop an incremental computing method, which can deal with complex modifications and therefore is suitable for abstract data structures. The key idea is to use shortcut fusion in order to decompose a complex modification to a series of simple ones. Based on this idea, we extend Jeuring's incremental computing method, which can deal with algebraic data structures, so as to deal with abstract data structures. Our method is purely functional and does not rely on any run-time support. Its correctness is straightforward from parametricity. Moreover, its cost is often proportional to that of the corresponding modification.

Submitted to:
International Symposium on Functional and Logic Programming (FLOPS) 2016.

コンセプト:
① 抽象データ構造に対するいくつかの処理に対し, 入力の変化に対する incremental computingを導出.

② 圏に基づいて手法を一般化.

アウトライン

1. 導入

2. スプレー木におけるincremental computing
3. 正規表現マッチングにおけるincremental computing

4. 一般のデータ型におけるincremental computing

Introduction

1. 導入

Incremental Computing

Incremental Computing

ある処理について, 入力が元のものから僅かに変化した際, 対応する新しい処理結果を一から再計算せず, **元の出力を利用して効率よく計算する**手法

Incremental Computing

x: 入力, f: 処理, modify: 変更

$$
\begin{aligned}
x &\mapsto f\ x \\
\mathrm{modify}\, x &\mapsto f(\mathrm{modify}\, x)
\end{aligned}
$$

通常の計算

Incremental Computing

x: 入力, f: 処理, modify: 変更, f_+: インクリメンタルな処理

$$x \mapsto f\ x$$
$$\mathrm{modify}\ x \mapsto f(\mathrm{modify}\ x)$$

通常の計算

$$x \mapsto f\ x$$
$$\mathrm{modify}\ x \mapsto f_+(f\ x)$$

f_+

incremental computing

Incremental Computing

x: 入力, f: 処理, modify: 変更, f_+: インクリメンタルな処理

$$\begin{aligned} x &\mapsto f\ x \\ \text{modify}\ x &\mapsto f\,(\text{modify}\ x) \end{aligned}$$

通常の計算

$$\begin{aligned} x &\mapsto f\ x \\ \text{modify}\ x &\mapsto f_+(f\ x) \end{aligned}$$

f_+

incremental computing

ただしf_+は $f\,(\text{modify}\ x) = f_+(f\ x)$ を満たす.

処理fと変更modifyに対しこのようなf_+を導出するのが目標.

実用例

エディタ

(例) HTMLソースの変更に対して高速にレンダリング

ビッグデータ分析

大量のデータに対してonlineに解析

エラーの復元

エラーを含むデータに対し、先に処理を行ってから
その結果に対し修正を行う

貢献

貢献: **抽象データ構造**を扱うプログラムに対し,
そのincremental computationを導出する手法を提案

ポイント

- 抽象データ構造は, よりシンプルな構造(e.g., 木, リスト)を組み合わせて作られる
- 抽象データ構造に対する変更は, よりシンプルな操作(e.g.,コンストラクタ, パターンマッチ)に分解できる

→ 変換操作の**多相化**, **shortcut fusion** を用いる

例: 集合

入力 s :: [a] 集合(ソート済みリスト)
処理 size :: [a] → Int 要素数の計算
変更 insert :: a → [a] → [a] 集合に値を追加

```
insert e s = case s of
  []    → [e]
  a:s′  → if e ≡ a then s
                   else if e < a then e:s
                                 else a:insert e s′
```

注意:

```
> insert 3 [1,2,4]
[1,2,3,4]
```

```
> insert 2 [1,2,4]
[1,2,4]
```

集合

目標: `size`の計算について, `insert e` による入力の変更に対応するincremental computing s_+ を求める

s_+ は `size (insert e s) = s+ (size e)` を満たす

注意: `size (insert e s)` は必ず `size s + 1` になるとは限らない. 前の注意参照.

多相化

insertをリストについて**多相化**した関数INSERTを考える.

```
insert e s = case s of
  []   → [e]
  a:s’ → if e ≡ a then s
                  else if e < a then e:s
                                 else a:insert e s’
```

```
INSERT :: (a, A → a → a) → (a → Maybe (A, a)) → A → a → a
INSERT (n, c) d e s = case d s of
  Nothing         → c e n
  Just (a, s') →
    if e ≡ a then s
         else if e < a then c e s
                       else c a (INSERT (n, c) d e s')
```

多相化

```
INSERT :: (a, A → a → a) → (a → Maybe (A, a)) → A → a → a
INSERT (n, c) d e s = case d s of
  Nothing      → c e n
  Just (a, s') → if e ≡ a then c e s
                          else c a (INSERT (n, c) d e s')
```

A	値の型
a :: * → *	抽象データ型，リストに相当
n :: a	データコンストラクタ，Nil([]) に相当
c :: A → a → a	データコンストラクタ，Cons(:) に相当
d :: a → Maybe (A, a)	具体的な処理
e :: (追加する)値	s :: (追加対象の)抽象データ構造

多相化

insertはINSERTを用いて表すことができる:

```
INSERT :: (a, A → a → a) → (a → Maybe (A, a)) → A → a → a
INSERT (n, c) d e s = case d s of
  Nothing        → c e n
  Just (a, s') → if e ≡ a then c e s
                            else c a (INSERT (n, c) d e s')
```

```
insert e s = INSERT
  ([], (:))
  (λl → case l of
          []  → Nothing
          a:x → Just(a, x))
```

Incremental Computing

s_+はINSERTを用いて書くことができる

```
INSERT :: (a, A → a → a) → (a → Maybe (A, a)) → A → a → a
INSERT (n, c) d e s = case d s of
  Nothing        → c e n
  Just (a, s') → if e ≡ a then c e s
                              else c a (INSERT (n, c) d e s')
```

```
s₊ = INSERT
  (0, λ_ x → x + 1)
  (λx → if x ≡ 0 then Nothing
                   else Just (a, x - 1))
```

Incremental Computing on Splay Trees

2. スプレー木における Incremental Computing

スプレー木

スプレー木 (Splay Tree)

二分探索木の一つ.

ノードを参照する際に, 木の**回転(スプレー操作)**を施し参照したノードを根に移動させることで, 以降の参照を高速に行うことができる.

二分木とスプレー操作

```
data BTree = Nd BTree Int BTree | Lf
```

```
splay k t = case t of
  Lf        → t
  Nd l v r →
    if k ≡ v
      then t
      else if k < v
             then case splay k l of
                    Lf           → t
                    Nd l' v' r' → Nd l' v' (Nd r' v r)
             else case splay k r of
                    Lf           → t
                    Nd l' v' r' → Nd (Nd l v' l') v' r'
```

k :: Int　参照する値　　t :: BTree　二分木

スプレー操作

```
> exampleTree
--8
  |--9
  |  |-- Lf
  |  `-- Lf
  `--6
     |--7
     |  |-- Lf
     |  `-- Lf
     `--2
        |--5
        |  |-- Lf
        |  `--4
        |     |-- Lf
        |     `--3
        |        |-- Lf
        |        `-- Lf
        `--1
           |-- Lf
           `-- Lf
```

```
> splay 3 exampleTree
--3
  |--8
  |  |--9
  |  |  |-- Lf
  |  |  `-- Lf
  |  `--6
  |     |--7
  |     |  |-- Lf
  |     |  `-- Lf
  |     `--5
  |        |-- Lf
  |        `--4
  |           |-- Lf
  |           `-- Lf
  `--2
     |--1
     |  |-- Lf
     |  `-- Lf
     `-- Lf
```

計算の例: 木の高さ

二分木の高さを計算する関数heightを考える.

入力	t	::	BTree	二分木
処理	height	::	BTree → Int	木の高さの計算
変更	splay k	::	BTree → BTree	kを根に回すスプレー操作

一般に, スプレー操作後の木の高さは再計算しないとわからない.

```
> height exampleTree
7
> map (height . flip splay exampleTree) [1..8]
[8,7,6,5,5,6,7,7]
```

木の高さのIncremental Computing

目標: 木 `t`について, スプレー操作後の木の高さ
`height (splay k t)` をインクリメンタルに計算する

すなわち,

`height (splay k t) =` height_+ `(height t)`

を満たすheight_+ `:: Int → Int` を見つければよい.

しかしそのままでは不可能なので問題を変換する(後述).

木の高さ(fold版)

二分木を対象とする計算はfoldで表現できる.

```
fold_BTree :: (Maybe (α, Int, α) → α) → BTree → α
fold_BTree φ Lf           = φ Nothing
fold_BTree φ (Nd l v r) = φ $ Just (fold φ l, v, fold φ r)
```

ϕ :: Maybe (α, Int, α) → α **述語**

```
height :: BTree → Int
height = fold_BTree φ_height
```

```
φ_height :: Maybe (Int, Int, Int) → Int
φ_height Nothing              = 1
φ_height (Just (l, _, r)) = 1 + max l r
```

Upward Accumulation

foldを用いて得た木の高さはただの数値なので，そのままでは incremental computingできない.

そこでJeuringの手法[1]に基づき，**ラベル付二分木**と**upward accumulation** [2,3]を導入する.

Upward Accumulation

ラベル付き二分木

```
data BTree = Nd BTree Int BTree | Lf
```

```
data LBTree a = LNd (LBTree a) Int (LBTree a) a | LLf a
```

ノードNdと葉Lfが型aのラベルを持つ.

Upward Accumulation

upward accumulation

与えられた述語ϕと二分木tについて, tをtの各subtreeにおけるϕの計算結果をラベルに持つ二分木に変換する

```
data LBTree a = LNd (LBTree a) Int (LBTree a) a | LLf a
```

```
ua :: (Maybe (α, Int, α) → α) → BTree → LBTree α
ua φ = fold_BTree φᴸ
```

```
φᴸ :: Maybe (LBTree t, Int, LBTree t) -> LBTree t
φᴸ Nothing = LLf (φ Nothing)
φᴸ (Just (l, v, r)) =
     LNd l v r $ (φ . Just) (val l, v, val r))
```

`val :: LBTree a → a` ラベル付き二分木の根ノードのラベルを得る

例: Upward Accumulation

```
> exampleTree
--8
  |--9
  |  |-- Lf
  |  `-- Lf
  `--6
     |--7
     |  |-- Lf
     |  `-- Lf
     `--2
        |--5
        |  |-- Lf
        |  `--4
        |     |-- Lf
        |     `--3
        |        |-- Lf
        |        `-- Lf
        `--1
           |-- Lf
           `-- Lf
```

val ∘ ua ϕ = fold ϕ

```
> ua ϕheight exampleTree
--8 [7]
  |--9 [2]
  |  |-- Lf [1]
  |  `-- Lf [1]
  `--6 [6]
     |--7 [2]
     |  |-- Lf [1]
     |  `-- Lf [1]
     `--2 [5]
        |--5 [4]
        |  |-- Lf [1]
        |  `--4 [3]
        |     |-- Lf [1]
        |     `--3 [2]
        |        |-- Lf [1]
        |        `-- Lf [1]
        `--1 [2]
           |-- Lf [1]
           `-- Lf [1]
```

ラベル[*] = 部分木の高さ

Incremental Computing

元のincremental computingの等式

$$\mathtt{height}\ (\mathtt{splay}\ \mathtt{k}\ \mathtt{t}) = \mathtt{height}_{+}\ (\mathtt{height}\ \mathtt{t})$$

の代わりに,

$$\mathtt{ua}\ \phi_{\mathtt{height}}\ (\mathtt{splay}\ \mathtt{k}\ \mathtt{t}) = \mathtt{height}_{+}\ (\mathtt{ua}\ \phi_{\mathtt{height}}\ \mathtt{t})$$

を満たす

$$\mathtt{height}_{+} :: \mathtt{LBTree} \rightarrow \mathtt{LBTree}$$

を求める問題として考えればよい.

多相化

Idea: `splay`関数の多相化

shortcut fusionの考え方を用いてsplayを多相化する.

```
splay :: Int → BTree → BTree
splay k t = ...
```

```
SPLAY :: (Maybe (α,Int,α) → α, α → Maybe (α,Int,α)) →
         Int → α → α
SPLAY (in, out) k t = ...
```

(i, o)は二分木のコンストラクタ・デストラクタ

多相化

スプレー操作 splay (再掲)

```
splay :: Int → BTree → BTree
splay k t = case t of
  Lf       → t
  Nd l v r →
    if k ≡ v
      then t
      else if k < v
        then case splay k l of
          Lf           → t
          Nd l' v' r' → Nd l' v' (Nd r' v r)
        else case splay k r of
          Lf           → t
          Nd l' v' r' → Nd (Nd l v' l') v' r'
```

多相化

splay操作を多相化したSPLAY

```
SPLAY :: Int → (Maybe (α, Int, α) ↔ α) → α → α
SPLAY k (in, out) t = case t of
  Nothing        → t
  Just (l v r) →
    if k ≡ v
      then t
      else if k < v
        then case out (SPLAY k (in, out) l) of
          Nothing          → t
          Just (l' v' r’) → in (Just (l’, v’, in (Just (r' v r))))
        else case out (SPLAY k (in, out) r) of
          Nothing          → t
          Just (l' v' r’) → in (Just (in (Just (l' v l)), v’, r’))
```

注意: (f, g) :: (A ↔ B) ≡ (A → B, B → A)

多相化

splayはSPLAYで書ける.

```
in_BTree :: Maybe (BTree, Int, BTree) → BTree
in_BTree Nothing            = Lf
in_BTree (Just (l, v, r)) = Nd l v r
```

```
out_BTree :: BTree → Maybe (BTree, Int, BTree)
out_BTree Lf          = Nothing
out_BTree (Nd l v r) = Just (l, v, r)
```

```
splay k :: BTree → BTree
SPLAY k :: (Maybe (α, Int, α) ↔ α) → α → α
```

```
splay k = SPLAY k (in_BTree, out_BTree)
```

Incremental Computing

スプレー操作に対応するincremental computing は SPLAYを用いて書くことができる.

まず関数 expose を導入する.

```
expose :: LBTree a → Maybe (LBTree a, Int, LBTree a)
expose (LLf _)         = Nothing
expose (LNd l v r _) = Just (l, v, r)
```

exposeはLBTreeの根ノードのみをdeconstructする

uaとexposeの性質

```
ua :: (Maybe (a, Int, a) → a) → BTree → LBTree a
ϕ  :: Maybe (a, Int, a) → a
ϕᴸ :: Maybe (LBTree a, Int, LBTree a) → LBTree a
expose :: LBTree a → Maybe (LBTree a, Int, LBTree a)
```

```
ua ϕ (Nd l v r) = ϕᴸ (Just (ua ϕ l, v, ua ϕ r))

expose (ua ϕ (Nd l v r)) = Just (ua ϕ l,v,ua ϕ r)
```

スプレー木のIncremental Computing

二分木t について, スプレー操作splayと計算fold ϕ に対応する incremental computing はSPLAY, ϕ^L, exposeを用いて書くことができる.

すなわち, 二分木を対象とする任意の計算fold ϕ について以下の等式が成り立つ:

$$\texttt{ua}\ \phi\ (\texttt{splay k t}) = \texttt{SPLAY k}\ (\phi^L, \texttt{expose})\ (\texttt{ua}\ \phi\ \texttt{t})$$

```
splay   :: Int → BTree → BTree
SPLAY   :: Int → (Maybe (α, Int, α) ↔ α) → α → α
ua      :: (Maybe (a, Int, a) → a) → BTree → LBTree a
ϕ       :: Maybe (a, Int, a) → a
ϕᴸ      :: Maybe (LBTree a, Int, LBTree a) → LBTree a
expose  :: LBTree a → Maybe (LBTree a, Int, LBTree a)
```

木の高さを求める

二分木tについて，スプレー操作splayと高さを求める関数heightに対応するincremental computing height_{+} は以下のように書ける.

```
splay :: Int → BTree → BTree
```

```
height :: BTree → Int
height = fold_BTree φ_height
```

$$\text{ua}\ \phi_{height}\ (\text{splay k t}) = \text{height}_{+}\ (\text{ua}\ \phi_{height}\ \text{t})$$

$$\text{ua}\ \phi_{height}\ (\text{splay k t}) = \text{SPLAY k}\ (\phi^{L}_{height},\ \text{expose})\ (\text{ua}\ \phi_{height}\ \text{t})$$

$$\phi^{L}_{height} :: \text{Maybe (LBTree Int, Int, LBTree Int)} \rightarrow \text{Int}$$

→ デモ

一般の木の変更

splay操作 `splay k :: BTree → BTree` に限らず,

任意の二分木に対する操作

```
modify :: BTree → BTree
```

について, 多相化した関数

```
MODIFY :: (Maybe (α, Int, α) ↔ α) → α → α
```

が存在すれば, incremental computing が得られる.

$$\boxed{\texttt{ua}\ \phi\ (\texttt{modify t}) = \texttt{MODIFY}\ (\phi^{L}, \texttt{expose})\ (\texttt{ua}\ \phi\ \texttt{t})}$$

Incremental Regular Expression Matching on Zippers

3. 正規表現マッチングにおける Incremental Computing

概要

Zipper構造で表された文字列

```
Zipper = (SList A, Clist B)
```

と基本操作

```
right (xs, Cons(y, ys))  = (Snoc(xs, y), ys)
left  (Snoc(xs, x), ys)  = (xs, Cons(x, ys))
remove (Snoc(xs, a), ys) = (xs, ys)
insert a (xs, ys)        = (Snoc (xs, a), ys)
```

で構成された変更に対して, Zipper上のfold演算で表されている正規表現マッチングをインクリメンタルに行う (詳細略)

Datatype-Generic Incremental Computing

4. 一般のデータ型における Incremental Computing

F-algebra

F-代数 (F-algebra)

自己関手Fについて, **F-algebra**とは

- 対象　　A
- 射　　　　　ϕ : FA → A

の組(A, ϕ).

(再帰的な)データ構造を表すことができる

Initial Algebra

F-始代数 (initial algebra)

F-代数 (μF, in_F)が**initaial**であるとは, 右下の図式を可換にする射 h が唯一存在することを言う.

唯一の射h: μF → A を

catamorphism または **fold**

と呼び, $[\![\phi]\!]_F$ で表す.

$$
\begin{tikzcd}
F\mu F \arrow[r, "Fh"] \arrow[d, "\mathrm{in}_F"] & FA \arrow[d, "\phi"] \\
\mu F \arrow[r, "h"] & A
\end{tikzcd}
$$

Initial Algebra

initial algebraの対象μFとFμFの間には同型射が存在する

(右図でA = FμFとすればよい)

したがって, μFはデータ型の等式 $X \cong FX$ の最小不動点.

$$
\begin{tikzcd}
F\mu F \arrow[r, "Fh"] \arrow[d, "\mathrm{in}_F"] & FA \arrow[d, "\phi"] \\
\mu F \arrow[r, "h"] & A
\end{tikzcd}
$$

in_F: FμF → μF に対して

out_F: μF → FμF

とする.

例: 二分木

二分木の型BTreeは始代数で表現できる

```
φ              :: Maybe (α, Int, α) → α
fold_BTree φ   :: BTree → α
in_BTree       :: Maybe (BTree, Int, BTree) → BTree
out_BTree      :: BTree → Maybe (BTree, Int, BTree)
```

Maybe (・, Int, ・) は 定数関手$!_A$と恒等関手1によって，関手

$$T = 1\times !_{\mathrm{Int}} \times 1 + !_{\{()\}}$$

で表すことができる.

よって始代数 $(\mu T, \mathrm{in}_T)$ が得られる. これがBTreeと対応している.

例: 二分木

`Maybe (・, Int, ・)`と関手T, `BTree`とμTがそれぞれ対応している

```
ϕ              :: Maybe (α, Int, α) → α
fold_BTree ϕ   :: BTree → α
in_BTree       :: Maybe (BTree, Int, BTree) → BTree
out_BTree      :: BTree → Maybe (BTree, Int, BTree)
```

$$
\begin{aligned}
\phi &: TA \to A \\
[\![\phi]\!]_F &: \mu T \to A \\
\mathrm{in}_T &: T\mu T \to T \\
\mathrm{out}_T &: T \to T\mu T
\end{aligned}
$$

$$T = 1 \times !_{\mathrm{Int}} \times 1 + !_{\{()\}}$$

Incremental Catamorphism

Idea: incremental computing (例: 二分木)をμFに一般化

Def1. Incremental Catamorphism

μF上の**incremental catamorphism**とは射のtriple

$$h : \mu F \to A$$
$$\phi : FA \to A$$
$$\psi : A \to FA$$

$$\begin{tikzcd} F(\mu F) \arrow[r, "Fh"] \arrow[d, "\mathrm{in}_F"'] & FA \arrow[d, "\phi"] \\ \mu F \arrow[r, "h"] & A \end{tikzcd} \qquad \begin{tikzcd} F(\mu F) \arrow[r, "Fh"] & FA \\ \mu F \arrow[r, "h"] \arrow[u, "\mathrm{out}_F"] & A \arrow[u, "\psi"'] \end{tikzcd}$$

で右の図式を可換にするもの.

Generic Upward Accumulation

二分木`BTree`について, ラベル付き木`LBTree α`を与えて upward accumulationを適用することで incremental computingを求めることができた.

一般のデータ構造Fについても同様に, 型Aの情報を加えた構造$F_A = F \times !_A$ を与えることで, upward accumulation を適用することができる.

詳細略

抽象データ構造

抽象データ構造(ADS; abstract data structure)

データ構造Tにおける**抽象データ構造(ADS)**とは,

- データ構造T
- 操作M ⊆ { f: T → T}

の組(T, M).

“抽象データ型（ちゅうしょうデータがた、英: abstract data type、ADT）とは、データ構造とそれを直接操作する手続きをまとめてデータ型の定義とすることでデータ抽象を実現する手法またはそのひとまとまりとして定義されたデータ型を言う”
(抽象データ型 – Wikipedia)

ADSに対する操作

データ $t \in \mu F$ に対する操作 $M \ni \mathrm{modify} : \mu F \to \mu F$ に対して, incremental computingできるか？

incrementalizable

$\mathrm{modify} : \mu F \to \mu F$ が**incrementalizable**であるとは,

$$\mathrm{MODIFY} : (F\alpha \to \alpha, \alpha \to F) \to \alpha \to \alpha$$

が存在し

$$\mathrm{modify}\ t = \mathrm{MODIFY}\ (\mathrm{in}_F, \mathrm{out}_F)\ t$$

を満たすことを言う.

ADSにおけるIncremental Computing

Theorem.

$(\mu F, M)$ をMの操作がincrementalizableなADS

(h, ϕ, ψ) をincremental catamorphism

とする. このとき以下の等式が成り立つ:

$$h \circ \mathrm{modify} = \mathrm{MODIFY}\ (\phi, \psi) \circ h$$

Future Work

“*our approach fails to deal with accumulative objective functions*“

“More recursion schemes are studied in the literature, including anamorphisms, hylomorphisms, adjoint folds/unfolds, and conjugate hylomorphisms. It would be interesting if our theory can be extended to these recursion schemes.”

所感

upward accumulationに依存している (各ラベルにデータを保存している) ので, 計算したい値が数値ではなく木構造そのものである場合などは空間効率が悪い