DLisp

Difference List Processor

`https://github.com/sano-jin/dlisp`

Jin SANO

October 17, 2021

1 概要

Inplace rewriting を用いたデータの操作は，破壊的な手法を用いることのできない場合と比較して，時間的にも空間的にも，効率よく出来ることの幅が広がるため，有用である．ただし，共有オブジェクトに対して破壊的更新をすると，プログラムの純粋性を担保できなくなる．所有権解析により，共有を行っていないことを静的に保証することも可能であるが，強い制約によりプログラミングの容易さを損なう可能性がある．純粋さを担保するために，破壊的更新を行う際は必ずオブジェクト全体を複製してから行うことにする方法も考えられるが，これはオブジェクトの大きさに比例したコストがかかり，非効率である．そこで，所有権解析に通らなかった共有される可能性のあるオブジェクトは，破壊的操作を試みる際に，操作の履歴を発行することで，過去の状態を復元できる手段を残しながら，最新のオブジェクトを手にいれるコストを最小化する手法を提案する．この手法が顕著に活かせる例として，アトムと差分リストのみからなる Lisp 派生言語 DLisp を実装した．DLisp はリスト末尾を破壊的に更新することで append が $O(1)$ で可能である（より正確にはアッカーマン関数の逆関数のオーダだが，これは多くの場合静的解析で削減可能と考えられる）．

2 導入

既存の言語では，基本的には「後で元のオブジェクトも使うかもしれないとき」は，事前にそのオブジェクト全体を複製する．特に，関数型言語，スクリプト系，ground を用いた LMNtal など．しかし，それにはオブジェクトの大きさに比例したコストがかかる．しかも，後で「使うかもしれない」ということは使わない可能性もある．

つまり，既存のパラダイムでは

1.「事前に莫大なコストを払い」,
2.「後で古いものを使うときのコストはゼロ」としていた．

しかし，基本的には **新しいものを使う可能性が高い**と考えられるので,

1.「事前のコストは最小化」した上で,
2. 後で「古いものを使いたくなったら（多少の）コストを払って戻す」ようにしたい．

そこで,

1.「後で元のオブジェクトも使うかもしれない」のに，破壊的更新を行うときは，「破壊的操作の履歴」をコミットすることにする
   - このコミットは単にアドレスとそこに代入した値のペアさえあればよく，「オブジェクト全体の複製」などよりも遥かに低コストである．
2. 新しいものを使い続ける場合は上記の履歴のコミット以外のコストは払わない
   - 最新であるかどうかのチェックはビットが立っているかどうかなどで $O(1)$ で行う
3. 古いものに戻したい場合は，「（多少の）コストを払って復元する」
   - 基本的に新しいものを使い続けることの方が多いはずなので，**古いものを欲しがる人にコストを払わせる**．
   - ただし，このコストはあくまで破壊的更新の回数（append の回数など）であり，純粋にできる部分は純粋に保っていたらそこまで大きくはならないはず．

3 提案手法

3.1 概要

破壊的更新の履歴を木構造で管理する．

欲しいオブジェクトに対応する葉と現時点で最新のオブジェクトに対応する葉の間のパスのノードにある操作（逆操作）を実行しながらこれらのノードを逆向きにつなぎ直す．

最古のノードから最新のオブジェクトに対応するノードまでのパスを Main stream と呼ぶことにする．Main stream から派生したパス（Main stream から取り残されてしまったブランチ）を Sub stream と呼ぶことにする．

- Master stream と Sub stream は区別できる必要がある（タグをつけておく）
- と思っていたが，その必要はないかも知れない．統一した方がより綺麗に実装できる．ただ，区別していた方が理解が容易である気もするのでとりあえずこのままにしておく．

オブジェクトが最新かどうかはそれが参照する履歴ノードが Master stream 上にあり，かつその次の履歴ノードが存在しない（Master stream の最新である）かで確認できる．

- より効率化するためにオブジェクト自体にフラグを用意しておくことも考えられる．

オブジェクトが最新でなかった場合は

- オブジェクトが参照する履歴ノードが Sub stream 上であった場合は
  1. Sub stream を上へたどっていき,
  2. LCA に到達したら Master stream を下へ（次へ）最新のオブジェクトに紐づいている履歴ノードまで辿っていき,
  3. Master stream 上の操作（逆操作）を帰りがけ順に実行しながら，履歴ノードに登録されている操作を履歴ノードに元々あった操作の逆操作（順操作）で更新して Sub stream のノードにする
  4. Sub stream 上の操作（逆操作）を帰りがけ順に実行しながら, 履歴ノードに登録されている操作を履歴ノードに元々あった操作の逆操作（順操作）で更新して Master stream のノードにする
- オブジェクトが参照する履歴ノードが Master stream 上であった場合は上記の 2 – 3 を実行する

というだけ（これ以上ないくらい非常にシンプル）
ただし，素朴な手法なので要改良である.

- 特に差分リストの場合はもっと最適化できる

3.2 実装

```
type history_node =
  | Main of
      (node ref * node) *
      history_node ref option
  | Sub of
      (node ref * node) *
      history_node ref
(* node は 差分リストのノード.
 * 現在履歴を管理するデータは差分リストのノードだけ
 *)
```

Listing 1: History node

```
(** Main stream を辿りながら帰りがけに逆実行する.
辿ってきた node は sub stream 化して逆順につなぐ.
*)
let rec traverse_main_stream parent_ref
    this_ref =
  match !this_ref with
  | Sub _ ->
      failwith @@ "substream should not be
          reached from main stream"
  | Main ((addr, value), next_ref_opt) ->
      (match next_ref_opt with
      | None -> ()
      | Some next_ref -> traverse_main_stream
          this_ref next_ref);
      let old_value = !addr in
      addr := value;
      this_ref := Sub ((addr, old_value),
          parent_ref)

(** 履歴を辿る.
+ Sub stream を上へ辿って行き,
+ Main stream (LCA) に辿り着いたら（ただし，LCA の操作は実行しない）, [traverse_main_stream ] を実行し,
+ その後帰りがけ順に sub stream を順実行しながらこれを main stream 化する.
*)
let rec traverse_history next_ref_opt this_ref
    =
  match !this_ref with
  | Sub ((addr, value), parent_ref) ->
      traverse_history (Some this_ref)
          parent_ref;
      let old_value = !addr in
      addr := value;
      this_ref := Main ((addr, old_value),
          next_ref_opt)
  | Main (addr_value, old_next_ref_opt) ->
      (match old_next_ref_opt with
      | None -> ()
      | Some old_next_ref ->
          traverse_main_stream this_ref
          old_next_ref);
      this_ref := Main (addr_value,
          next_ref_opt)

(** 差分リストを評価する前にはこの関数を実行して,
履歴を辿って差分リストを最新の状態にし，履歴を更新する必要がある.
この関数のみ外部に公開しておけば良い.
*)
let update = traverse_history None
```

Listing 2: Update

これだけ.
しかも，Main stream と Sub stream の区別がいらないなら，コード行は更にこの半分以下になる.

4 例題

```
;; ++ は append

(let ((x '(1 2 3)))
  (let ((y (++ x '(4 5 6))))
    (let ((z (++ x '(7 8 9))))
      (let ((w (++ y '(10 11 12))))
        (begin
         (print x)
         (print y)
         (print z)
         (print w)
         (print x)
         (print y)
         (print z)
         (print w)
         )
        )
      )
    )
  )
```

Listing 3: Append

これの実行結果が

```
(1 2 3)
(1 2 3 4 5 6)
(1 2 3 7 8 9)
(1 2 3 4 5 6 10 11 12)
(1 2 3)
```

```
(1 2 3 4 5 6)
(1 2 3 7 8 9)
(1 2 3 4 5 6 10 11 12)
(1 2 3 4 5 6 10 11 12)
```

こうなる

- print は引数を評価してそれを標準出力に表示して，引数の値を返す built-in 関数

5 課題

5.1 Occur checking

自分自身を含むオブジェクトを連結してしまうと，循環してしまうため（素朴な手法では）評価が無限ループしてしまう．

```
(let ((x '(1 2 3)))
  (let ((y (++ x x)))
    (begin
     (print x)
     (print y)
     )
    )
  )
```

Listing 4: Circle

これを防ぐためには連結の前に Occur checking，つまり連結しようとしているオブジェクトに「重なり」がないかをチェックしてやれば良い．

Occur checking の（素朴な）実装は Union-find を用いれば良い．差分リストの id（オブジェクトのアドレスを用いれば良い）で素集合データ構造を作ってやれば，アッカーマン関数の逆関数のオーダで自分自身を含む差分リストを連結しようとしてないかが判別できる．本実装ではそのようになっている．もちろん理想的には静的に所有権解析などを行うことで，この動的な手間はほとんどの例で削減が可能と思われる．

- より理想的には SMT solver などを用いて等号論理を解くものと思われる

自分自身を含む差分リストを連結しようとしている場合は，残念ながら従来の append を行う他ないと思われる．

- が，そもそも自分自身を連結するコードをユーザがそんな頻繁に書くとは思えない．同じものをたくさん並べることは基本的には無意味なので．

5.2 その他最適化手法

- 現実装は，すごく安直で，全ての操作を逆実行する（Nil（未具現化変数）の更新（具現化）をした部分もわざわざ戻す）が，差分リストの場合は最適化が可能
  - 現実装は一般のグラフへの適用を考えた（差分リストに最適化されていない）素朴な手法

6 関連研究

6.1 Multiversion concurrency control

複数の書き込みに対して一貫性を保つという意味で本手法とは逆の関係にあると考える．提案手法では，データの一貫性を保つのではなく，それぞれが自分の持っているバージョンにアクセスできるようにする．

6.2 可逆プログラミング

Janus 低レベルでグラフのようなデータ構造を扱うことまで頭が回っていないように見える

6.3 Haskell の DList など

Haskell などの関数型言語には，差分リストを「シミュレート」するための手法が存在している．このように，関数型言語ではデータ構造とその操作をめちゃめちゃ工夫して「償却時間は」$O(1)$ のアルゴリズムを発明する傾向にある

- `https://hackage.haskell.org/package/dlist`
- `https://wiki.haskell.org/Difference_list`
- `https://okmij.org/ftp/Haskell/zseq.pdf`
- ただし，あくまで「償却時間」であり，毎回それが保証されるわけではない
- 「工夫」はあまり自明でない
- 「工夫」のために無駄な中間データ構造を要求する場合が多い
  - 定数倍で性能が悪化
  - メモリの消費（メモリ消費は GC のタイミングを早めるため速度にも影響するはず）

高階関数を用いた差分リストの場合は，

- Thunk が大量発生するのでメモリ効率が悪い
- 正格の場合は head をとるために $O(n)$ かかる（と思うのだがもっと調べる必要がある）
- いづれにせよ，破壊的な接続よりも効率的だとは思えない
- また，これらの手法は決して自明ではない（一般のグラフへの拡張は少なくとも著者には無理に思える）

7 まとめ

共有されたオブジェクトの破壊的操作を試みる際に，事前にオブジェクト全体を複製するのではなく，操作の履歴を発行することで，過去の状態を復元できる手段を残しながら，最新のオブジェクトを手にいれるコストを最小化する手法を提案する．

この手法が顕著に活かせる例として，アトムと差分リストのみからなる Lisp 派生言語 DLisp を実装した．DLisp は append が $O(1)$ で可能である（より正確にはアッカーマン関数の逆関数）．