DLisp

Differential List Processor

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

Jin SANO

October 11, 2021

Contents

TODO

- 提案書
  - 関連研究を載せる
  - 参考文献を書く（org-mode での書き方がよくわからない）
  - 提案手法を説明する図を追加する
  - モチベをもっとクリアにする
- 実装
  - car, cdr, cons などの builtin 関数
  - Occur checking
  - 実装の容易さのために一度 OCaml のリストに変換してから評価しているので，差分リストのまま扱うようにする
  - 履歴の最適化（の検討）

1 概要

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（リストの長さではなく）append の回数のアッカーマン関数の逆関数だが，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

2.1 背景

従来のプログラミング言語におけるデータの持ち方の考え方

C/C++ などの古典的な手続き型言語 破壊的更新が標準

- `const` が明示的に記述されていないと破壊的更新の可能性を否定できない
- 共有されているオブジェクトでもお構いなしに破壊的更新を行う
- どうしても「元のオブジェクト」が欲しくなる場合は，ユーザが明示的にその実装を行う必要がある
  - 再帰的にコピーを行う関数を実装するものと思われる

Rust などの比較的新しい手続き型言語 破壊的更新を行うが，所有権解析により共有されている可能性を排除することもできる

- そもそもオブジェクトの共有自体が推奨されない
- 実行効率は良いし安全性も高いが，制限が強すぎて，一部の低レイヤ好きにしか好まれない

Python/JavaScript などの比較的新しいスクリプト言語 関数型の思想を言語仕様・ライブラリなどでどんどん取り入れている

- 基本的に実行効率はお構いなし
- ガンガン複製する

関数型言語 immutable が標準

- append などを行う際は複製を行う必要がある
- そもそも append などのコストがかかる作業を避けるプログラムを意識的に書く傾向にある
- データ構造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）

論理型言語 未具現化変数

- append は一回しかできない
- 一度のみ具現化可能というのは Rust の所有権解析と少し似ている

LMNtal link による強い制約

- link は 2 頂点を繋ぐことしかできない
- hyperlink を用いてデータを共有することもできるが，共有物を純粋に保つのが面倒で（共有物を純粋に保たないプログラミングが人類に可能だとは思えない），静的な検査もないため，めちゃめちゃバグりやすい
- 結局 ground を使うことになる
  - C などのように再帰を回してオブジェクトを clone するコードを自前で書かなくて良いというメリットはある
  - が，とてもコストが大きい
  - のにめちゃめちゃ頻繁に使われている

まとめると，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

- 関数型言語，スクリプト系，LMNtal (with ground), …

しかし，それにはとてもコストがかかる

- オブジェクトの大きさに比例したコスト

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

2.2 先行研究

要調査

2.2.1 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（要調査）

2.2.2 Haskell の DList や高階関数を用いた手法

- 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（グラフへの拡張を考えたときに不利なはず）

2.3 本稿の構成

todo

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

occur checking はまだ実装していないので

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

Listing 4: Circle

このように自分自身を含む差分リストを連結できてしまい，その場合はグラフが循環するため，評価しようとすると無限ループし，`Stack_overflow` する．

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．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．

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．

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 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）．

7 メモ

差分リストはリストよりも強力なデータ構造

- append が $O(1)$ でできる
- 他の操作はリストと同等

ただし，差分リストの append

8 2021/10/10

提案手法は，単に共有物に対して破壊的操作を行っている場合は履歴を保持するというだけ．

- つまり，これはグラフに限らず，例えば配列などに対しても適用可能ではある．
- ただし，配列はめちゃめちゃ破壊的操作を行うため，履歴が大量発生する＆戻すのに操作の数だけ逆操作するため，あんまり嬉しくはない．
- （単方向の）差分リスト（もどき）が嬉しいのは，「末尾の破壊的更新」しかできないということであった．
  - 末尾の破壊的更新以外は純粋にできるため，それらの履歴の保持が不要であり，「履歴のコストが比較的小さい」というメリットがあった．
  - これは先週の段階ではぼんやりとしか理解していなかった（ので説明ができなかった）
  - こう言った性質をグラフ（の shapetype のような型）において自動的に導出できるのかは不明．
  - 双方向リストにしてしまうと，Head に cons するだけでも破壊的更新をする必要があり，この履歴も管理せざるを得なくなるため，履歴のコストが無視できなくなる（かも）
  - 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，こういった部分に関しても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

  - 差分リストの場合は move 可能な部分は履歴を管理する必要がないというのが僕の直感的な理解（あまりきちんと説明できないのでちゃんと例題を書く必要がある）
    * ただし，一般のデータ構造に対してはこれは保証できないことに気づいた（配列の破壊的代入など）

まとめると，

- 提案手法が差分リスト（もどき）において有効なのはほぼ確信している．
  - これをあまり理解してもらえなかったのは純粋に説明が悪かったのだと思う．
- 提案手法がより一般のグラフにおいて最適化可能なのかはよくわからない．
  - 操作の数だけ復元にコストがかかる可能性がある．
- が，仮に最適化できなかったとしても，「純粋（風に）にグラフ（破壊的データ構造）を扱う」という「今まで人類ができなかったこと（調べ学習が足りていない感はある）」を実現しているのでこれは価値があると思っている．
  - つまり，「今までできていたことをより良くする研究」ではないということ．
- 提案手法が一般にはコストゼロで途中の move と組み合わせられない（move できる部分は履歴の保持がいらないというのは一般には保証できない）のは痛手であったが，所有権解析を取り入れている言語は「徹頭徹尾」move させるようにしているので，途中の move があまり最適でないというのは仕方のないことだと言える．
  - ただし，差分リストの場合はこれがおそらく可能で，どういうパターンのときにそうなるのかはもっと考える必要がある（考える価値があると思う）

（木だけではなく）グラフ（特に差分リスト）を扱うメリット：

- グラフを扱えるとより時間効率の良い実装ができる場合がある．
  - キューなど
  - 末尾再帰化してループへ変換できる関数がある．
- グラフを扱えるとより空間効率の良い実装ができる場合がある．
  - 従来の append は第一引数のリストを複製するため，その分メモリを消費する．
  - 末尾再帰化可能でない関数はスタックを消費する．
- グラフを扱えるとより直感的に記述できる可能性がある．
  - キュー．2 本スタックを用意するのはすごく直感的というわけではない．

そもそも所有権解析により，常に move させるようにするのではダメなのか？

- 常に move させるのでは困るという明確な例題は正直あまり思い付いていない．
- だが，例えば python ユーザに所有権解析を押し付けるのはどうかと思う．
  - という非形式的な感想しかないと言われればそこまで．
- alim さんの ground を用いた hypergraph による lambda も ground を用いていて，しかもラムダ式は木にちょっと毛が生えたくらいだから提案手法で（効率的に）扱えないと困る．
  - この場合，ground のように丸々コピーするよりも遥かに安価である（あって欲しい）
  - しかし，そもそも lambda のエンコードの価値が実はあまりよくわからないから，これが最適化されることの意味もよくわからない．