Design and implementation of a pure functional language with GDT (Graph Data Type)

Jin SANO

September 27, 2021

Contents

1 Introduction

Functional languages feature only tree (ADT) as a data.

- `let rec` can create a cyclic data, but it is limited and cannot be pattern-matched as a graph.
  - i.e. it (just) creates an infinite graph in a user point of view
  - (<-> implementation. Especially, the virtual machine).

However, we often desire of using a more powerful data structures (e.g. Queue) for the efficiency in the execution of a program or to write program easily. Thus, we introduced GDT, graph data type, which is an extension of the former ADT (algebraic data type).

There are two main difficulties on achieving this.

1. How should we **implement a pure with graph** as a primary data structure (or can't we?).
2. How can we apply the **type system** (or can't we?)

1.1 The design choice we made in GDT (graph data type)

There are many choices in designing this.

1. Is Directed or Not-directed?
2. Is rooted?
3. Is DAG only?
4. Is hyperlink allowed?
5. Is (the incoming) link ordered?
    - For example, should we distinguish the graph which is a. Node `A` and `B` are pointing `C`. i.e. `(A(x), B(x), x -> c)`. b. Node `B` and `A` are pointing `C`. i.e. `(B(x), A(x), x -> C)`.

I chose the following graph AT THIS TIME (I possibly change my mind later. The final goal may be the (hyper)lmntal graph).

1. Directed (since ADT (=tree) is directed)
2. Rooted.
3. Only DAG is allowed in the pattern matching.
    - This is JUST A DESIGN CHOICE.
        - We will possibly extend it later.
    - Note cycle is still allowed in the generated graph (i.e. GDT is a super set of ADT)
4. Hypergraph (if we disallow hyperlinks, we can only treat tree, since the incoming link is allowed to write once for each node due to the constraint below)
5. Incoming link is not ordered.
    - It is allowed to write once for each node. e.g. `x -> P(...)`.
    - c.f. ADT (= tree) has only one (unordered) incoming link.

1.2 Possible usecases of GDT

Our primary motivative example is a Queue (FOR NOW).

- Queue should be able to add (and possibly access and remove) an element to the last of it with cost $O(1)$.

There are two ways to implement this in the former functional languages.

1. We can implement this using two stacks (lists).
    - It anables to add/access/remove the last element in $O(1)$, if we amortize the costs.
    - However, it cannot always sure the cost ($O(1)$).
    - and it generates intermediate data (we need two stacks. in other words, we should pay DOUBLE cost).
2. We can implmenent this in IMPERATIVE manner (with side-effects).
    - This is not pure and is not welcomed.
        - side-effects makes things difficult for a programmer.
        - side-effects makes it difficult to read a program.
        - side-effects makes it difficult to test (Unit test, Integration test).

GDT let us to define and use a $O(1)$ and pure queue, which was not possible in the former languages (not just the functinal languages but ALL).

The other usecase is a skip-list. (I am looking for other examples. Please help me)