

# DEPTH FIRST SEARCH ANTARA FUNGSIONAL DAN IMPERATIF

SUYANTO

Fakultas Matematika Dan Ilmu Pengetahuan Alam  
Jurusan Matematika  
Universitas Sumatera Utara

## Abstrak

Beberapa sifat dasar dari pemrograman secara fungsional seperti ketersediaan fungsi-fungsi berorde tinggi (higher-order functions) dan sifat lazy evaluation, menjanjikan pendekatan baru dalam rekayasa perangkat lunak dengan banyak kemudahan baik dalam rase disain dan implementasi (rapid proto typing) maupun dalam fuse pengetesan dan pelacakan kesalahan yang pada akhirnya akan mengurangi biaya pemrograman. Namun demikian ada pendekatan dasar yang berbeda yang digunakan pada pemrograman fungsional dibandingkan pemrograman imperatif yang sudah memberikan banyak kontribusi pada perkembangan dan kemajuan aplikasi komputer seperti yang sudah kita rasakan hingga sekarang ini. Perbedaan tersebut bukan hanya perbedaan pada sintaks pemrograman tetapi lebih dari itu adalah perbedaan dalam kerangka berfikir. Dengan mencoba menerapkan algoritma graph, lebih khususnya DFS pada kedua pendekatan tersebut, paper ini bermaksud untuk memberikan beberapa arahan dan perspektif pada para pemrogram yang berasal dari komunitas imperatif untuk dapat juga melakukan elaborasi dari kemampuan bahasa fungsional.

## 1. Pendahuluan

Pemrograman Fungsional disebut demikian karena memang dalam sebuah program seluruhnya hanya terdiri dari fungsi-fungsi. Program utama itu sendiri ditulis sebagai fungsi yang menerima masukan program sebagai argument dan menghasilkan keluaran program sebagai result. Biasanya program utama didefinisikan dalam fungsi-fungsi berorde tinggi, dan fungsi-fungsi tersebut didefinisikan dalam fungsi-fungsi lain yang mempunyai orde lebih rendah, demikian seterusnya hingga pada level yang paling rendah yaitu language primitives.

Karakteristik dan keuntungan pemrograman fungsional sering dirangkum sebagai berikut. Pemrograman fungsional tidak mengandung pernyataan assignments, sehingga variabel-variabelnya, sekali diberi harga tidak akan pernah berubah. Dengan demikian secara umum pemrograman fungsional sama-sekali tidak mengandung side-effects. Suatu pemanggilan fungsi tidak mempunyai efek selain penghitungan hasilnya. Hal ini mengeliminasi penyebab utama adanya bugs dan semua hal yang membuat urutan pengekseskusion menjadi tidak relevan karena tidak pernah ada side-effect yang dapat merubah harga dari suatu ekspresi, sehingga dapat dievaluasi kapan saja kita mati. Hal ini membebaskan pemrogram dari beban untuk mendefinisikan flow of control.

Komunitas pemrograman fungsional terbagi dalam dua kelompok yakni bahasa-bahasa yang murni (pure) seperti Miranda dan Haskel, yang sederhana dan menerapkan kalkulus lambda murni dan bahasa-bahasa yang tidak-murni (Impure) seperti Scheme, Standard ML, yang menerapkan kalkulus lambda yang sudah diiringi dengan beberapa side-effects, seperti assignment, exceptions, atau continuation, dimana bahasa-bahasa yang tidak murni ini menawarkan keuntungan efisiensi dan terkadang membuat ekspresi yang lebih kompak.

Perkembangan terakhir dibidang komputasi teoritis, khususnya dibidang type theory dan category theory, telah menyarankan pendekatan baru yang memungkinkan untuk mengintegrasikan kelebihan-kelebihan dari bahasa murni dan tidak murni yaitu dengan menggunakan monads ; yang mengintegrasikan efek-efek impure kedalam bahasa fungsional yang pure.

Gofer adalah environment( dengan kata lain, sebuah interpreter) untuk pemrograman pure fungsional yang pada prinsipnya sama dan dalam beberapa hal berusaha meniru Haskel compiler. Tetapi tidak seperti Haskel yang komersial, Gofer merupakan program publik. Penerapan DFS fungsional dalam paper ini ditulis dalam Gofer dan DFS imperatif ditulis dengan C.

## 2. Algoritma Graph

Algoritma graph tidak mempunyai sejarah yang menguntungkan dalam bahasa fungsional yang pure. Belum semuanya terjelaskan seperti misalnya bagaimana cara untuk mengekspresikan algoritma tanpa menggunakan side-effects untuk meningkatkan efisiensi dan bahasa yang bersifat lazy ,yang secara alami menghindari side-effects tersebut. Menurut Launcburry, representasi graph yang sudah dilakukan orang secara murni terkadang lebih buruk dibandingkan dengan representasinya dalam secara imperatif tradisional, baik dalam ukuran graph (menjadi kuadratik) maupun compactness dari program. Oleh karena itu ia menawarkan pendekatan baru dalam penerapan algoritma graph secara fungsional dimana pendekatan tersebut dipakai sebagai acuan pada paper ini.

Perhatian kita pusatkan pada algoritma DFS. Algoritma DFS pertama kali diperkenalkan oleh Tarjan dan Hopcroft 20 tahun yang lalu. Mereka menunjukkan bagaimana DFS dapat digunakan untuk mengkonstruksikan sejournal algoritma graph yang efisien. Dalam prakteknya hal ini dilakukan dengan menempelkan code fragments yang dibutuhkan untuk algoritma tersebut kedalam prosedur DFS sehingga dapat kita dapat menghasilkan informasi yang relevan ketika penelusuran/pencarian (search) berlangsung. Walaupun pendekatan imperatif tradisional tersebut diatas cukup elegan ternyata pendekatan tersebut mempunyai beberapa kelemahan. Pertama, kode program DFS jalin-menjalin (intertwined) dengan kode program dari algoritma yang graph tertentu (rnisalnya colouring/pewarnaan edges) sehingga program menjadi monolitik. Kode program tidak dibangun dengan komponen-komponen yang dapat digunakan kembali (re-use), dan tidak ada pemisahan antara rase-rase yang sebenarnya secara logis terpisah. Kedua, untuk dapat menurunkan algoritma DFS, kita harus mengasumsikan suatu proses dinamik apa dan kapan sesuatu terjadi dan pemikiran seperti itu adalah sangat kompleks.

Kadang-kadang, DFS forest diperkenalkan dalam rangka untuk menyediakan sebuah harga static untuk membantu penalaran. Kita membangunnya diatas ide tersebut. Jika kita mempunyai suatu DFS forest yang eksplisit, adalah membantu penalaran kita, maka selama overheads masih dapat diterima, hat tersebut baik untuk pemrograman. Pada paper ini kita menyajikan beberapa algoritma DFS sebagai kombinasi dari komponen-komponen standar, melewati harga-harga intermediate dari satu fungsi ke fungsi lain secara eksplisit, dan hasilnya adalah tingkat modularitas yang lebih baik dibanding algoritma tradisional.

Tentu saja, ide untuk memisah algoritma menjadi beberapa fase terpisah yang dihubungkan oleh struktur data menengah (intermediate data structure) bukan hal yang baru. Dan dalam beberapa hal malah memang terjadi di semua paradigma pemrograman, khususnya pemrograman fungsional. Yang baru adalah, penerapan hat tersebut untuk algoritma graph. Tantangannya adalah bagaimana menemukan suatu harga menengah (intermediate value) yang fleksibel.

## 2.1 Representasi Graph Adjacency Lists

Kita merepresentasikan graph sebagai immutable array, yang diindeks dengan verteks-verteksnya, dimana setiap komponen dari array adalah sebuah list dari verteks-verteks yang 'reachable' sepanjang edge tunggal. Hal ini memberikan waktu pengaksesan yang konstan.

Graph dapat dipandang sebagai sebuah tabel yang diindeks oleh verteks-verteks.

```
type Table a = Array Vertex a
type Graph   = Table [Vertex]
```

Tipe dari Vertex dapat berupa sebarang tipe yang dipunyai oleh kelas indeks `!x` dari Gofer, seperti `Int`, `Char`, tuples of indices, dan lain-lain. Untuk saat ini kita akan mengasumsikan :

```
type Vertex = Char
```

Kita akan membuat penyederhanaan asumsi bahwa verteks-verteks dari graph adalah tipe kontinyu (misalnya angka dari 0 -59, atau karakter dari 'a' sampai 'z' dan lain-lain). Jika tidak maka suatu fungsi hash harus dipakai untuk pemetaan. Karena kita mengasumsikan kontinuitas, biasanya kita akan merepresentasikan list of vertices dengan sebuah pasangan titik tepi :

```
type Bounds = (Vertex, Vertex)
```

array pada Gofer dilengkapi dengan operator pengindeksan (`!`) dan fungsi-fungsi : indices (mengembalikan sebuah list dari indeks-indeks) dan bounds (yang mengembalikan sebuah pasangan indeks terkecil dan indeks terbesar).

Untuk memanipulasi tabel-tabel (termasuk graphs) kita mendefinisikan suatu fungsi numerik generik yaitu `mapT` yang mengaplikasikan argumen fungsinya pada setiap tabel pasangan indeks/entry, dan menciptakan tabel baru.

```
mapT :: (Vertex -> a -> b) -> Table a -> Table b
mapT f t = array (bounds t) [v := f v (t!v) | v <- indices t]
```

fungsi `array` dari Gofer mengambil tepi-tepi atas dan bawah dan sebuah list dari pasangan indeks/harga, dan membangun array darinya dalam waktu yang linier.

Akhirnya, kadang-kadang berguna untuk menterjemahkan sebuah list berurutan dalam look-tip-table yang memperlihatkan posisi dari vertex dalam list. Untuk keperluan kita dapat menggunakan fungsi `tabulate`:

```
Tabulate :: Bounds -> [Vertex] -> Table Int
Tabulate bnds vs = array bnds (zip vs [ i ..])
```

yang meringkas verteks-verteks bersama dengan bilangan bulat positif 1,2,3,... dan membangun sebuah array dari angka-angka tersebut (dalam waktu tinier), yang diindeks oleh verteks-verteksnya.

## 2.2. Edges

Kadang-kadang lebih mudah untuk mengekstrak sebuah list of edges dari suatu graph. Sebuah edge adalah sepasang verteks. Tetapi, karena beberapa graph

adalah sparse, kita juga perlu mengetahui pasangan-pasangan verteks itu secara terpisah.

```
type VE      = (Bounds, [Verteks : = Verteks ])
edges :: Graph -> VE
edges g = (bounds g, [ v:=w | v <- indices g, w <- g!v])
```

Untuk membangun sebuah graph dari sebuah list of edges kita mendefinisikan sebuah fungsi

```
BuildG.
BuildG :: VE -> Graph
BuildG (bnds,s) = accumArray snoc [] bnds es
where snoc xs x = x : xs
```

### 2.3 Operasi-operasi sederhana

Untuk menemukan immediate successors dari sebuah verteks dalam sebuah graph  $g$  kita secara sederhana akan menghitung  $g ! v$ , yang mengembalikan a list of vertices.

#### Transpose graph

Menggabungkan fungsi edges dan buildG memberikan kita cara untuk membalik semua edges dalam sebuah graph dengan fungsi transposeG :

```
transposeG :: Graph -> Graph
transposeG g      = buildG (vs, (w:=v <- es])
  where (vs,es) = edges g
```

kita mengekstrak edges dari graph orisinal, membalik arahnya dan membangun kembali graph dengan edges yang baru.

#### OutDegree dan InDegree

Menggunakan mapT kita dapat mendefinisikan,

```
outdegree :: Graph -> Table Int
outdegree g = mapT numEdges g
  where numEdges v ws = length ws
```

yang membangun sebuah tabel yang terdiri dari jumlah edges yang keluar dari sebuah verteks.

Dengan menggunakan transposeG kita dapat secara cepat mendefinisikan sebuah tabel indegree untuk verteks-verteks :

```
indegree :: Graph -> Table Int
```

```
lindegree g = outdegree (transposeG g)
```

### 3. DFS

Cara tradisional memandang DFS sebagai suatu proses yang secara lepas dapat digambarkan sebagai berikut. Mula-mula, semua verteks dari graph dinyatakan sebagai "unvisited", maka kita memilih satu dan mengeksplorasi sebuah edge yang mengantarkan kepada verteks baru. Sekarang kita mulai pada verteks tersebut dan mengeksplorasi sebuah edge yang mengantarkan kepada verteks baru lagi. Kita meneruskan proses ini dengan cara tersebut sampai kita mencapai sebuah

verteks yang tidak punya edges yang mengantarkan kita ke verteks-verteks lain yang belum dikunjungi. Pada saat ini kita backtrack dan meneruskan dari verteks terakhir yang mengantarkan kepada verteks yang belum dikunjungi.

Akhirnya kita akan mencapai sebuah titik dimana setiap verteks yang 'reachable' dari verteks awal sudah dikunjungi semua. Jika masih ada verteks yang tersisa, kita pilih satu dan memulai pencarian lagi, sampai akhirnya setiap verteks sudah dikunjungi sekali, dan setiap edge sudah diuji.

Pada paper ini yang mengacu kepada pekerjaan Launchbury, kita mengkonsentrasikan pada DFS yang lebih sebagai sebuah spesifikasi untuk sebuah harga (value) daripada sebagai sebuah proses. Harga yang dispesifikasi adalah spanning forest yang didefinisikan oleh sebuah DF traversal dari sebuah graph, yakni sebuah sub-graph tertentu dari graph aslinya yang ketika mengandung semua verteks, biasanya menghilangkan banyak edges.

### 3.1 Spesifikasi dari DFS

Sebagai pendekatan pada algoritma DFS yang kita eksplor dalam paper ini adalah untuk memanipulasi depth-first forest secara eksplisit, tahap pertama adalah mengkonstruksi depth-first forest dari sebuah graph. Untuk melakukan hal tersebut kita membutuhkan definisi trees dan forests.

Sebuah forest adalah a lists of trees, dan sebuah trees adalah sebuah node yang mengandung beberapa value, bersamaan dengan sebuah forest dari sub-trees. Baik trees maupun forests keduanya polimorphic dalam tipe dari data yang boleh mereka kandung.

Data Tree a = Node a (Forest a)

type Forest a = [Tree a]

Sebuah DFS dari sebuah graph mengambil suatu graph dan sebuah urutan awal verteks-verteks. Semua verteks dalam graph dalam urutan awalnya akan menjadi the return forest.

dfs :: Graph -> [Vertex] -> Forest Vertex

fungsi ini merupakan inti dari bahasan ini.

### 3.2. Implementasi DFS

Dalam rangka menerjemahkan sebuah graph menjadi sebuah depth-first spanning tree kita menggunakan suatu teknik umum yang digunakan dalam pemrograman fungsional yang bersifat lazy: generate then prune. Diberikan sebuah graph dan a list of vertices, pertama kita akan men-generate sebuah forest (mungkin saja infinite) yang terdiri dari semua verteks dan edges dalam suatu graph, dan kemudian prune (forest yang telah kita buat) untuk menghilangkan pengulangan.

#### 3.2.1. Generating

Kita mendefinisikan fungsi generate yang, jika diberikan suatu graph g dan sebuah verteks v akan membangun sebuah tree yang root nya adalah v yang mengandung semua verteks dalam g yang 'reachable' dari v.

generate :: Graph -> Vertex -> Tree Vertex

generate g v = Node v (map (generate g) (g!v))

Jika graph  $g$  adalah siklik, maka tree yang dibangkitkan akan infinite. Tentu saja karena tree dibangkitkan on demand, hanya seporsi tertentu yang akan dibangkitkan. Bagian yang di prune tidak akan pernah dikonstruksi.

### 3.2.2. Pruning

Tujuan utama dari proses pruning adalah untuk mengabaikan subtrees yang memiliki akar-akarnya (roots) telah terkonstruksi sebelumnya. Oleh karena itu kita perlu me-maintain sekumpulan verteks yang jumlahnya terhingga (secara tradisional disebut dengan 'marks ') dan verteks-verteks yang akan diabaikan.

Cara yang termudah untuk mencapai hal ini adalah dengan menggunakan state transformers, dan menirukan teknik imperatif yang me-maintain suatu array of booleans, yang diindeks oleh sekumpulan elemen. Inilah yang kita lakukan. Implementasi dan sekumpulan verteks adalah sebagai berikut :

```
type Set s = MutArr s Vertex Bool
mkEmpty :: Bounds -> ST s (Set s)
mkempty bnds = newArr bnds False
```

```
contains :: Set s -> Vertex -> ST s Bool
contains m v = readArr m v
```

```
include :: Set s -> Vertex -> ST s ()
include m v = writeArr m v True
```

Sebuah set disajikan sebagai sebuah mutable array, diindeks oleh verteks-verteks, yang mengandung nilai-nilai boolean. Untuk men-generate sebuah set terhingga kita mengalokasikan suatu ukuran array tertentu dengan semua elemen kita inialisasi dengan nilai False. Keanggotaan dari Set dapat diatur hanya dengan menggunakan prosedur pembacaan dan penulisan array.

Prune didefinisikan sebagai berikut :

```
prune :: Bounds -> Forest Vertex -> Forest Vertex
rune bnds ts = runST (mkEmpty bnds 'thenST' \m ->
  chop m ts)
```

fungsi prune dimulai dengan memperkenalkan sebuah thread dari state yg baru, kemudian men-generates sebuah array kosong didalam thread tersebut dan memanggil 'prosedur' chop. Hasil akhir dari prune adalah harga yang dibangkitkan oleh chop, yaitu state akhir yang diabaikan.

```
chop :: Set s -> Forest Vertex -> Forest Vertex
chop m [] = returnST []
chop m (Node v ts : us)
  = contains m v 'thenST' \visited ->
    if visited then
      chop m us
    else
      include m v          'thenST' \_ ->
      chop m ts            'thenST' \as ->
      chop m us            'thenST' \bs ->
      returnST ((Node vas) : bs)
```

Ketika melakukan chopping pada a list of trees, rootnya yang pertama diuji. Jika sudah ada sebelumnya, seluruh tree tersebut akan diabaikan. Kalau tidak akan ditambah pada suatu set yang direpresentasikan oleh m, dan dua pemanggilan 'chop' berikutnya dibuat dalam sekuen.

Yang pertama, disebut chop m ts, melakukan prunes thd forest yang merupakan akarnya adalah v, menambahkan semua forest tersebut pada set dari vektors yang sudah ditandai. Setelah semuanya selesai, sub-forest yang hendak di prune diberi nama as, dan sisanya adalah forest aslinya yang di chop. Hasilnya diberi nama dengan bs, dan sebuah forest dikonstruksikan dari kedua list tersebut.

Semua ini dikerjakan secara lazy, on demand. State combinators memaksa komputasi untuk mengikuti urutan atau sequence yang sudah ditentukan sebelumnya. Pada titik ini kita mungkin heran, bagaimana pemrograman fungsional dapat dikatakan lebih menguntungkan. Karena ternyata, code nya terlihat mirip dengan imperatif dalam beberapa hal komentar yang demikian itu mungkin ada benarnya, namun perlu dicatat bahwa hal ini adalah satu-satunya tempat dalam fase pengembangan dimana kita melakukan operasi destruktif untuk meningkatkan efisiensi. Tambahan lagi, enkapsulasi secara sempurna telah disediakan oleh runST yang menjamin bahwa dfs mempunyai eksterior fungsional murni.

Komponen-komponen dari generate dan prune kemudian digabungkan untuk menghasilkan definisi drTi depth-first search.

Dfs f vs = prune (bounds g) (map (generate g) vs)

argumen vs adalah a list of vertices, sehingga fungsi generate di map ke vs. Forest yang dihasilkan kemudian di prune secara preorder.

### 3.3 Kompleksitas

Model untuk analisis kompleksitas dari program imperatif standar (tradisional) menyatakan bahwa algoritma DFS adalah linier dalam ukuran graphnya (oleh karena itu, dapat dieksekusi dalam  $O(V+E)$ )

Model yang bersesuaian dengan itu pada pemrograman fungsional pada algoritma yang ditawarkan oleh Launchbury juga linier dan waktu pengeksekusiannya juga konstan namun diperkirakan akan kehilangan sebuah faktor kira-kira 6, jika dibandingkan dengan implementasinya dalam bahasa C.

## 4. Analisa

Imperatif	Fungsional
<ul style="list-style-type: none"> <li>Aliran parameter fungsi terlihat secara eksplisit (baik passing by value maupun by reference).</li> </ul>	<ul style="list-style-type: none"> <li>Aliran parameter fungsi terlihat secara implisit yakni tersembunyi dibalik definisi tipe dari fungsi tersebut.</li> </ul>
<ul style="list-style-type: none"> <li>Pemrograman tidak harus selalu menggunakan fungsi standar (seringkali kita dapat membuat user-defined function) sehingga dalam banyak hal kita tidak perlu memahami definisi fungsi standar</li> </ul>	<ul style="list-style-type: none"> <li>Karena semua fungsi didefinisikan oleh fungsi lain yang mempunyai orde lebih rendah, akan lebih nyaman bila kita mengerti mendefinisikan fungsi standar (yakni fungsi primitif yang sudah disediakan)</li> </ul>
<ul style="list-style-type: none"> <li>Sangat mungkin masih terdapat bugs (runtime errors) setelah program selesai</li> </ul>	<ul style="list-style-type: none"> <li>Karena tidak diizinkan adanya variabel side effect kita dapat memastikan bahwa bugs</li> </ul>

dikodekan karena variable – variable side effect seperti counter, error detection dan continuation merupakan bagian penting dari program	(yakni run-times error yang disebabkan oleh side effects)tidak akan terjadi. Namun demikian hal tersebut tidak membebaskan kita dari kesalahan dalam mendisain program.
<ul style="list-style-type: none"> <li>Program lebih sulit dibaca dan difahami karena abstraksinya berorde lebih rendah (program ditulis dengan subtil – subtil program yang menghalangi pemahaman pembaca sehingga seringkali harus disertai dengan komentar yang cukup panjang)</li> </ul>	<ul style="list-style-type: none"> <li>Program lebih mudah dibaca dan difahami karena abstraksinya berorde lebih tinggi (dari kode program saja maksud dari program karena subtil program sudah dijelaskan pada fungsi – fungsi yang berorde lebih rendah).</li> </ul>
<ul style="list-style-type: none"> <li>Dari segi ukuran , LOC (Lines of code) lebih banyak</li> </ul>	<ul style="list-style-type: none"> <li>Dari segi ukuran, LOC (Lines of code) lebih sedikit</li> </ul>

## 5. KESIMPULAN DAN SARAN

### KESIMPULAN

Pada paper ini telah di coba mengimplementasikan DFS dalam dua pendekatan yaitu imperatif dan fungsional. Ada beberapa hal yang perlu dicatat :

1. Pemrograman fungsional pada dasarnya menawarkan suatu cara disain dan implementasi yang lebih bersifat alami / natural yang lebih conform dengan cara berfikir manusia, namun demikian ada batasan-batasan dalam bahasa pemrograman fungsional terutama yang pure (seperti dihindarinya side-effect dan lazy evaluation) harus betul-betul difahami terlebih dahulu.
2. Pada implementasi yang mengharuskan adanya side-effect maka hal itu dapat disimulasikan dengan menggunakan monad.

### SARAN

Beberapa saran yang ingin disampaikan pada paper ini:

1. Akan terasa lebih mudah jika kita membandingkan pemrograman imperatif dan fungsional dengan memulainya pada implementasi sederhana.
2. Mungkin akan ada gunanya jika kita mencoba menyelesaikan suatu masalah, khususnya DFS ini dengan membandingkan bukan saja antara fungsional dan imperatif tapi juga antarpure dan impure languages (antara Gofer dengan Scheme misalnya).
3. Paper ini dapat dilanjutkan dengan mengadakan pengujian lebih seksama terhadap kompleksitas baik waktu maupun kapasitas penyimpanan.

Diharapkan paper ini dapat menjadi pengantar yang cukup dalam bagi para pemrogram dan peneliti yang tertarik dengan pemrograman fungsional.

## Daftar Pustaka

Thon Launchburry, Graph Algorithm with a Functional Flavour, Oregon Graduate Institute.

Philips Wadler, Monads for functional programming, University of Glasgow, Scotland.

John Hughes, Why functional Programming Matters, Institutionen for Datavetenskap, Sweden.

Mark P. Jones, GOFER functional programming environment, version 2.20 - 2.30, 1995.