KOMBINATOR-Y
UNTUK MELAKUKAN MEMOIZING FUNGSI REKURSIF

Subandijo

Computer Science Department, School of Computer Science, Binus University
Jln. K. H. Syahdan No. 9 Palmerah Jakarta Barat 11480
subandijo1030@gmail.com

ABSTRACT

This article discusses how to find a solution of a recursive function such as Fibonacci or factorial numbers without repetition. Therefore, a recursive function is considered a fixed-point of non-recursive function. To calculate the fixed-point, we can use Y Combinator, a non-recursive functions to perform memoizing recursive function. This method can significantly reduce the execution time of recursive functions.

***Keywords**: memorizing, recursive function, fixed-point, Y combinator, javascript*

ABSTRAK

Artikel ini membahas bagaimana mencari solusi suatu fungsi rekursif seperti bilangan Fibonacci atau faktorial tanpa menggunakan perulangan. Untuk itu fungsi rekursif dipandang sebagai fixed-point suatu fungsi non-rekursif. Untuk menghitung fixed-point, kita dapat menggunakan Kombinator-Y, suatu fungsi non-rekursif untuk melakukan memoizing fungsi rekursif. Metode ini secara signifikan bisa mengurangi waktu eksekusi fungsi rekursif.

***Kata kunci:** memoizing, fungsi rekursif, fixed-point, kombinator-y, javascript*

PENDAHULUAN

Setiap fungsi rekursif terdiri dari dua bagian, yaitu *base case* dan *reduction step*. *Base case* digunakan untuk mengakhiri pemanggilan fungsi, dengan tujuan untuk menghindari terjadinya *infinite looping*. Sedangkan *reduction step* digunakan untuk mengurangi nilai parameter. Pengurangan nilai parameter dilakukan sampai ketemu *base case*. Contoh klasik adalah sekuen bilangan Fibonacci: Fib(0) = 0 dan Fib(1) = 1 adalah *base case* sedangkan Fib(n) = Fib(n-1) + Fib(n-2) adalah *reduction step*. Setiap kali definisi rekursif dieksekusi, nilai *n* dikurangi dengan 1 atau 2 sampai didapat *n* = 0 atau *n* = 1.

Fungsi rekursif umumnya lebih sederhana dan lebih mudah dipahami daripada iterasi karena sebagian dari beban yang menjadi tanggung jawab pemrogram dialihkan ke sistem. Dengan alasan ini penyelesaian suatu masalah menggunakan pendekatan rekursif banyak disukai oleh seorang pemula karena kesederhanaanya. Tetapi hal ini membutuhkan konsekuensi lain yaitu eksekusi fungsi rekursif jauh lebih lambat dan membutuhkan memori yang lebih besar daripada iterasi. Hal ini disebabkan karena hasil pemanggilan fungsi tidak langsung dihitung tetapi ditumpuk terlebih dulu di *stack* sebelum nantinya dihitung setelah *base case* dicapai.

Pendekatan lain yang lebih menjanjikan adalah, fungsi rekursif disajikan sebagai *fixed-point* yang mengijinkan digunakannya pendekatan *memoizing* kombinator *fixed-point*. Pendekatan kombinator ke rekursif memungkinkan pemanggilan internal ke fungsi rekursif dapat dilakukan secara otomatis sehingga tidak membebani waktu eksekusi. Sebagai ilustrasi pandang implementasi fungsi rekursif untuk menghitung bilangan Fibonacci di bawah ini. Kode tersebut adalah *tree-rekursive* dengan waktu eksekusi *exponential time*. Menggunakan *memoizing* kode ini dapat dimodifikasi tanpa meninggalkan sifat kemudahan untuk dibaca dengan waktu eksekusi *linear time.* Sekedar ilustrasi, perbandingan waktu eksekusi kedua metode untuk berbagai nilai *n* dapat dilihat di samping kode.

```
unsigned long Fib(int n) {
        if (n<=2) return 1
        else return (Fib(n-1) + Fib(n-2));
}
```

n	Fib(n)	Rekursif	Memoizing
30	83204	0.2000	0.0008
	35	92274764.2000	0.0008
	40	102334115 48.200	0.0010

Secara eksplisit masalah yang dibahas artikel ini adalah sangat lambatnya waktu eksekusi suatu fungsi rekursif sehingga diperlukan adanya suatu pendekatan baru. Dengan demikian tujuan dari penulisan artikel ini adalah untuk menemukan suatu metode yang dapat digunakan untuk mengurangi waktu eksekusi fungsi rekursif secara signifikan. Metode tersebut adalah Kombinator-Y. Artikel ini membahas bagaimana Kombinator-Y melakukan memoizing fungsi rekursif.

METODE

Memoizing

Di komputasi, *memoizing* kerap ditulis sebagai memorization, yang diartikan sebagai metode yang digunakan untuk meningkatkan kecepatan program komputer dengan cara menghindari pengulangan kalkulasi hasil sebelumnya pada saat dilakukan pemanggilan fungsi (Hoffman, 1992). *Memoizing* menggunakan teknik *caching* yang bisa memperbaiki kinerja *software* yang menggunakan pengulangan kalkulasi. Di sejumlah bahasa pemrogaman fungsional, *memoizing* dikenal dengan nama *tabling* atau *lookup table*. Istilah *memoizing* berasal dari kata Latin *memorandum* dan

pertama kali dikenalkan oleh Michie (1968). Pandang fungsi berikut untuk menghitung factorial bilangan *n*:

```
function faktorial (n adalah integer non-negatif)
        if n = then return 1
        else return faktorial (n-1) * n
        end if
end function
```

Versi *non-memoizing* di atas, yang melibatkan algoritma rekursif, akan memerlukan *n*+1 pemanggilan fungsi sebelum sampai pada hasil akhir dan setiap pemanggilan akan memerlukan *time cost*. Tergantung pada karakteristik mesinnya, *time cost* adalah jumlah dari komponen *cost* untuk kegiatan berikut: membentuk kerangka *stack* pemanggilan fungsi, membandingkan *n* dengan 0, mengurangi *n* dengan 1, membentuk kerangka *stack* pemanggilan fungsi berikutnya, mengalikan hasil pemanggilan fungsi dengan *n* dan menyimpan nilai balik sehingga nantinya dapat digunakan kembali jika diperlukan.
Fungsi faktorial versi *memoizing* adalah sebagai berikut:

```
function faktorial ( n adalah integer non-negatif)
        if n di dalam lookup-table then return lookup-table-value-for-n
        else if n = 0 then return 1
        else    let x = factorial (n-1) * n
                store x in lookup-table in the n position
                return x
        end if
end function
```

Dalam contoh ini, jika faktorial pertama kali dipanggil dengan nilai 5 dan kemudian dipanggil lagi dengan nilai lebih kecil atau sama dengan lima, nilai baliknya juga akan di-*memoizing* karena faktorial dipanggil secara sekursif dengan nilai 5, 4, 3, 2, 1, dan 0 dan nilai balik untuk setiap pemanggilan juga akan disimpan. Jika kemudian fungsi dipanggil lagi dengan nilai yang lebih besar daripada 5, 7 misalnya, hanya 2 pemanggilan rekursif yang dilakukan (7 dan 6) sebab nilai 5! telah disimpan pada pemanggilan sebelumnya. Dengan cara ini *memoizing* memungkinkan fungsi menjadi lebih efisien waktu eksekusinya. Makin kerap dipanggil makin lebih efisien yang pada akhirnya berujung pada kecepatan yang meningkat.

Fungsi Rekursif

Rózsa Péter (aslinya Politzer) adalah ibu dari teori fungsi rekursif (Shoenfield, 2004). Ia mempunyai banyak kontribusi dalam pengembangan teori matematika tetapi namanya kurang dikenal dibandingkan dengan Godel, Turing, Churh dan Kleene. Ia menyajikan makalah tentang fungsi rekursif di International Congress of Mathematicians di Zurich tahun 1932, di mana untuk pertama kalinya ia mengusulkan agar fungsi rekursif dipelajari secara terpisah sebagai subbidang dari matematika. Sebelumnya Godel menggunakan fungsi rekurif hanya sebagai alat untuk mempelajari *incompleteness*. "Mathematics is Beautiful," adalah judul kuliah terbuka yang diberikan Peter kepada guru dan siswa sekolah menengah dan diterbitkan dalam jurnal *Mathematik in der Schule 2* tahun 1964. Terjemahannya dalam bahasa Inggris dilakukan oleh Leon Harkleroad dari Cornell University dan diterbitkan di The Mathematical Intelligencer 12 tahun 1990 halaman 58-64.

Di komputasi, rekursif adalah metode di mana solusi suatu masalah tergantung pada obyek yang lebih kecil dalam masalah yang sama. Pendekatan ini dapat digunakan untuk banyak tipe masalah dan merupakan salah satu gagasan utama di komputasi. Kekuatan dari rekursif terletak pada kemungkinan untuk mendefinisikan himpunan obyek tidak berhingga dengan pernyataan yang

berhingga. Sebagian besar bahasa pemrograman mendukung rekursif dengan mengijinkan fungsi untuk memanggil dirinya sendiri. Sejumlah bahasa pemrograman fungsional tidak mendefinisikan konstruksi *looping* tetapi hanya menyajikan rekursif untuk mengulangi pemanggilan kode. Teori komputasi telah membuktikan bahwa bahasa yang hanya mengenal rekursif secara matematis setara dengan bahasa imperatif, dalam arti mereka bisa menyelesaikan masalah yang sama meskipun tanpa struktur kontrol seperti "*while*" dan "*for*".

Rekursif Sebagai Fixed-Point

Mahasiswa yang mengambil mata kuliah kalkulus sudah lama mengenal istilah rekursif dan *fixed- point* hanya saja mereka tidak pernah menyadarinya. Persamaan $x = x^2 - 2$ dipandang sebagai fungsi rekursif oleh pemrogram karena *x* didefinisikan sebagai suku dirinya sendiri.

Jika ditanya untuk menyelesaikan persamaan tersebut untuk mencari nilai *x,* ia akan diubah menjadi persamaan kuadrat. Akan tetapi ada cara lain yang dapat dilakukan untuk menyatakan dan mencari nilai *x* yaitu *fixed-point*.

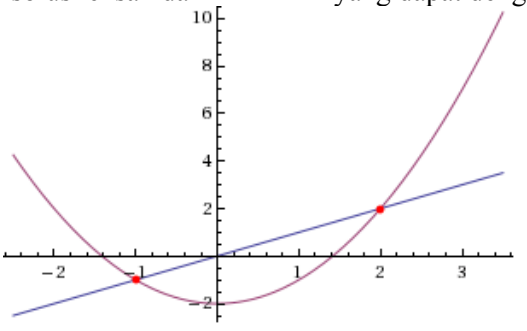
Ada banyak teori *fixed-point* tetapi yang paling terkenal adalah teori *fixed-point* L. E. J. Brouwer, seorang pakar matematika asal Belanda. Secara sederhana beliau menyatakan bahwa *fixed-point* dari suatu fungsi *f* adalah masukan yang sama dengan keluarannya, yaitu *x* adalah *fixed- point* dari fungsi *f* jika $x = f(x)$. Sebagai contoh 0 dan 1 adalah *fixed-point* dari fungsi $f(x) = x^2$ sebab $0 = 0^2$ dan $1 = 1^2$. *Fixed-point* fungsi order-satu, fungsi integer misalnya, adalah suatu nilai dengan order-satu sedangkan *fixed-point* fungsi order-tinggi *f* adalah fungsi lain *p* sedemikian hingga $p = f(p)$. Sejumlah fungsi tidak mempunyai *fixed-point* sedangkan sejumlah fungsi lain mempunyai banyak *fixed-point*. Notasi Fix(f) digunakan untuk menyatakan himpunan *fixed-point* fungsi *f*.

Definisikan fungsi $f(x) = x^2 - 2$. Kemudian amati bahwa persamaan awal dapat ditulis kembali dalam bentuk " $x = f(x)$ ". Dengan kata lain solusi dari persamaan tersebut adalah *fixed-point* dari fungsi *f*, yaitu $Fix(f) = \{-1,2\}$ yang dapat diverifikasi sebagai berikut

$$f(-1) = (-1)^2 - 2 = 1 - 2 = -1$$
$$f(2) = (2)^2 - 2 = 4 - 2 = 2$$

atau menggunakan grafik $y = x$ dan $y = f(x)$

Gambar 1 di bawah ini adalah solusi eksak dari $x = x^2 - 2$ yang dapat dengan mudah dihitung.



Gambar 1. Solusi eksak dari $x = x^2 - 2$.

Secara sepintas kekuatan dari teknik berikut adalah obvervasi yang menyatakan bahwa setiap saat kita punya definisi fungsi rekursif dalam bentuk "$x = f(x)$" maka *x* didefinisikan sebagai *fixed-point*. Triknya adalah mencari jalan untuk menemukan *fixed-point* jika persamaan berbentuk "$f = F(f)$

” , di mana nilai *f* bukanlah bilangan tetapi fungsi. Kombinator-Y dapat digunakan untuk menemukan *fixed-point*.

Kombinator-Y

Kombinator *fixed-point*, kerap ditulis kombinator *fixpoint*, adalah fungsi order tinggi yang menghitung *fixed-point* fungsi lain. Dengan demikian kombinator *fixed-point* adalah fungsi *g* yang menghasilkan *fixed-point* p untuk sembarang fungsi *f* sebagai berikut:

$$g(f) = p, \text{ di mana } p = f(p)$$

atau

$$g(f) = f(g(f)).$$

Karena kombinator *fixed-point* adalah fungsi order tinggi maka secara historis ia berhubungan langsung dengan pengembangan λ-kalkulus. Dalam penelitiannya tentang λ-kalkulus dan *combinatoric logic* Haskel B. Curry menemukan paradoks kombinator *fixed-point* yang disebutnya sebagai Kombinator-Y. Kombinator ini merupakan kombinator *fixed-point* yang paling terkenal dan mungkin paling sederhana Nash (2007).
Secara formal Kombinator-Y didefinisikan sebagai

$$Y = \lambda f.(\lambda x.f\ (\ x\ x))\ (\lambda x.f\ (x\ x))$$

Dapat dibuktikan fungsi ini adalah kombinator *fixed-point* dengan cara mengaplikasikannya ke fungsi *g* sebagai berikut:

$$\begin{aligned} Y\ g &= (\ \lambda f.(\lambda x.f\ (\ x\ x))\ (\lambda x.f\ (x\ x)))\ g \\ &= (\lambda x.\ g\ (x\ x))\ \ (\lambda x.g(x\ x)) \\ &= (\lambda y.\ g\ (y\ y))\ (\lambda x.\ g(x\ x)) \\ &= g\ (((\lambda x.\ g(x\ x))\ (\lambda x.\ g\ (x\ x))) \\ &= g\ (Y\ g) \end{aligned}$$

Kombinator-Y membutuhkan fungsional sebagai masukan dan mengembalikan *fixed-point* unik dari fungsional tersebut. Fungsional adalah fungsi yang memerlukan fungsi sebagai masukan. Dengan demikian *fixed-point* fungsional merupakan fungsi.

Menggunakan konsep fungsional dan *fixed-point* kita dapat mengeliminasi rekursif suatu fungsi melalui dua langkah yaitu pertama cari fungsional yang *fixed-point* nya adalah fungsi rekursif yang kita cari dan kedua cari *fixed-point* suatu fungsional tanpa rekursif Langkah pertama hanya memerlukan transformasi sederhana sedangkan Kombinator-Y akan menangani langkah kedua.

Derivasi Kombinator-Y

λ-kalkulus, bahasa di mana Kombinator-Y biasanya disajikan, adalah bahasa pemrograman yang hanya memuat fungsi anonim, fungsi aplikasi dan variabel referensi. Kombinator-Y memungkinkan fungsi anonim untuk memanggil dirinya sendiri, sehingga ia memungkinkan terjadinya fungsi rekursif anonim. Karena fungsi anonim tidak punya nama mereka tidak dapat merujuk dirinya sendiri dengan mudah.

Kombinator-Y adalah konstruksi yang bisa membantu mereka untuk merujuk dirinya sendiri. Di sejumlah bahasa yang mendukung penggunaan fungsi anonim, kombinator *fixed-point* dapat digunakan untuk mendukung penggunaan fungsi rekursif anonim yaitu ketidakharusan adanya pengikatan fungsi tersebut ke *identifier*. Dalam kasus ini, kombinator *fixed-point* kadang-kadang disebut rekursif anonim.
Notasi $\lambda v = e$ dibaca sebagai fungsi λ yang memetakan masukan *v* menjadi keluaran *e*. *JavaScript* mendukung penggunaan fungsi anonim sebagai berikut:

$$\lambda v = e \ \ == \text{function (v) \{ return (e); \}}$$

Sehingga jika kita bisa menemukan cara untuk menyajikan Kombinator-Y dalam λ-kalkulus, kita dapat menyajikannya pula dalam *JavaScript* (Stefanov, 2010).
Untuk menderivasi Kombinator-*Y*, mulailah dengan sifat-sifat inti masalahnya. Jika diberikan Kombinatory-*Y* suatu fungsional *F, Y*(*F*) haruslah berbentuk *fixed-point* sebagai berikut:

$$Y(F) = F(Y(F))$$

Translasi definisi ini dalam *JavaScript* adalah:

```
function Y(F) { return F(Y(F)); }
```

Jika fungsi ini dieksekusi jelas tidak akan berjalan karena fungsi *Y* akan memanggil dirinya sendiri yang berarti rekursif tak berhingga.
Menggunakan notasi λ-kalkulus kita menulis pemanggilan *Y* dalam bentuk:

$$Y(F) = F(\lambda\, x.(Y(F))(x))$$

Selanjutnya jika kita memanggil fungsi Y, ia akan langsung memanggil fungsi *F* dan mengirim parameter (λ x.(Y(F))(x)) yang setara dengan *fixed-point*. Pernyataannya di *JavaScript* adalah sebagai berikut:

```
function Y(F) { return F(function (x) { return (Y(F))(x) ; }) ; }
```

Aktualnya fungsi akan mencari *fixed-point* dari fungsional dan kita dapat menggunakannya untuk mengeliminasi rekursif. Tentu saja, seperti tertulis pada formula, fungsi *Y* memanggil dirinya sendiri secara rekursif, sehingga faktanya kita tidak mengeliminasi rekursif. Kita hanya memindahkan semuanya ke dalam fungsi *Y*.
Menggunakan konstruksi lain yang disebut Kombinator-U kita dapat mengeliminasi pemanggilan rekursif didalam Kombinator-Y menggunakan transformasi yang lebih banyak sebagai berikut:

$$Y = (\lambda h.\lambda F.F(\lambda\, x.((h(h))(F))(x)))\ (\lambda h.\lambda F.F(\lambda\, x.((h(h))(F))(x)))$$

Dalam formula ini sisi bagian kiri sama sekali tidak memuat referensi ke *Y*.

Implementasi Kombinator-Y

Sembarang bahasa tanpa tipe yang mengijinkan penggunaan fungsi anonim seperti JavaScript dapat digunakan untuk menyajikan Kombinator-Y tanpa rekursif, iterasi dan efek samping. Tanpa harus memahami bagaimana Kombinator-Y bekerja, kita tetap dapat melihat aksinya dan menverifikasi bahwa tidak ada rekursi dan iterasi yang digunakan. Contoh berikut adalah penggalan kode bagaimana menyajikan fungsi faktorial tanpa rekursif.

```
// Bagian pertama
var Y = function (F) {
 return (function (x) {
  return F(function (y) { return (x(x))(y);});
  })
          (function (x) {
  return F(function (y) { return (x(x))(y);});
  }) ;
} ;

// Bagian kedua

var FactGen = function (fact) {
 return (function(n) {
 return ((n == 0) ? 1: (n*fact(n-1))) ;
 });
} ;
```

Ada dua catatan yang perlu ditambahkan untuk lebih memahami dua penggalan kode di atas. Catatan di bagian pertama adalah suatu fungsional adalah fungsi yang mengambil fungsi lain sebagai input. Kombinator-Y mencari fixed-point fungsional dikirim sebagai argumen sehingga ia memenuhi sifat Y(F) = F(Y(F)). Catatan di bagian kedua adalah semua fungsi di atas adalah anonim. FactGen adalah fungsional yang fixed-point nya adalah faktorial., sehingga jika mengirim fungsi faktorial ke FactGen kita akan menerima kembali fungsi faktorial. Karena Kombinator-Y mengembalikan fixed-point suatu fungsional, aplikasi Kombinator-Y ke FactGen akan mengembalikan fungsi faktorial. Perlu dicatat juga bahwa Y dan FactGen tidak merujuk dirinya sendiri.

Definisi Kombinator-*Y* hanya memuat tiga tipe ekspresi yaitu: fungsi anonim, variabel referensi dan fungsi aplikasi. Setiap fungsi anonim mempunyai bentuk *function (argumen) {return ekspresi;}*. Kombinator-Y adalah ekspresi tertutup dalam arti tidak ada referensi eksplisit ke variabel di luar atau ke dirinya sendiri. Dengan demikian tidak ada rekursi, iterasi atau mutasi. Kombinator-*Y* memungkinkan terjadinya transformasi dari fungsi rekursif ke fungsi non- rekursif. Jika kita mempunyai fungsi rekursif *f* sebagai berikut:

```
function f (arg) {
... f ...
}
```

Definisi ini dapat ditransformasi ke bentuk non-rekursif

```
var f = Y(function(g) { return (function (arg) {
 ... g ...
 }) ;}) ;
```

HASIL DAN PEMBAHASAN

Memoizing Kombinator-Y

Kombinator-Y adalah hasil signifikan dari pengembangan teori komputasi dan teori bahasa pemrograman. Ia menawarkan pendekatan lain untuk memahami fungsi *nontrivial* dalam bentuk *fixed-point*, bukannya paradigma baku rekursif dan iterasi.

Selanjutnya, andaikan kita mendefinisikan fungsi rekursif menggunakan paradigma fungsional-*fixed-point*. Pertanyaannya, bisakah kita menbentuk kombinator *fixed-point* yang secara otomatis memberikan kinerja yang lebih baik bagi fungsi tersebut? Jawabannya ya bisa. Kita bisa membentuk *memoizing* kombinator *fixed-point*: kombinator yang serupa dengan Kombinator-Y untuk melakukan (*caching*) hasil sementara pemanggilan fungsi.

Sebagai contoh, definisi baku fungsi rekursif Fibonacci yang memuat dua pemanggilan rekursif sehingga kompleksitas waktunya eksponensial adalah sebagai berikut:

```
function fib(n) {
 if (n == 0) return 0 ;
 if (n == 1) return 1 ;
 return fib(n-1) + fib(n-2) ;
 }
```

Kita bisa mendefinisikan ulang fungsi Fibonacci menggunakan Kombinator-*Y* sebagai berikut:

```
var fib = Y(function (g) { return (function (n) {
 if (n == 0) return 0 ;
 if (n == 1) return 1 ;
 return g(n-1) + g(n-2) ;
 }) ; }) ;
```

Formulasi ini masih tetap mempunyai kompleksitas eksponensial, tetapi kita bisa mengubahnya menjadi waktu linear hanya dengan mengubah kombinator *fixed-point dengan membentuk Ymem*, *memoizing* Kombinator-Y, yang bertugas untuk menjaga hasil komputasi yang di-*cache*, dan mengembalikan hasil pre-komputasi jika dibutuhkan. Penggalan kode berikut adalah hasil modifikasi Matthew Might yang ditulis dalam blog pribadinya.

```
function Ymem(F, cache) {
    if (!cache)
        cache = {} ;                        // bentuk cache baru
    return function(arg) {
        if (cache[arg])
            return cache[arg] ;             // jawaban dalm cache.
        var answer = (F(function(n){
            return (Ymem(F,cache))(n);
        }))(arg) ;                           // hitung jawaban
        cache[arg] = answer ;                // cache jawaban
        return answer ;
    } ;
}

var fib = Ymem(function (g) { return (function (n) {
    if (n == 0) return 0 ;
    if (n == 1) return 1 ;
    return g(n-1) + g(n-2) ;
}) ; }) ;

document.getElementById("result2").innerHTML =  fib(100) ;
```

Fungsi utama *Ymem* adalah menerima fungsional (opsional) *cache* hasil serta mengembalikan *fixed-point* fungsional yang merupakan *cache* hasil sementara. Untuk itu ada dua tugas yang ditangani oleh *Ymem* yaitu pertama *Ymem* hanya bekerja untuk fungsi dengan satu argumen tapi hal ini dapat diatasi dengan aplikasi *JavaScript* dan menggunakan *tree-like cache* dan kedua Ymem hanya bekerja untuk nilai argumen yang dapat diberi indeks seperti bilangan dan *string*, tetapi dapat dikembangkan dengan memasok nilai argumen komparator sehingga dapat menggunakan *sorted tree* untuk *cache*.

Hasil akhir penggalan kode ini adalah bilangan Fibonacci ke-100 sebesar 354224848179262000000 yang dihitung secara instan sedangkan versi aslinya membutuhkan waktu yang diestimasi melampui umur dari alam semesta.

PENUTUP

Manfaat umum dari *memoizing* adalah ia menawarkan reduksi waktu eksekusi dan potensi memperbaiki kinerja secara keseluruhan tergantung pada apa yang ingin kita capai. Secara khusus is bermanfaat untuk mendeteksi aras rekursif yang ada pada program aplikasi yang kita rancang. Secara umum ia bekerja sangat baik jika hasil dari pengiriman himpunan parameter dan nilai ke fungsi akan selalu menghasilkan nilai yang sama yaitu masukan yang sama dengan keluarannya. Ia tidak dapat diaplikasikan untuk kasus di luar itu. Selain itu ia juga sangat bermanfaat untuk menggambar bentuk yang sangat kompleks seperti pohon dan *fractal* di kanvas sehingga utilitasnya kerap kali melampaui imaginasi kita.

Aplikasi praktis dari teori ini adalah fungsi rekursif yang disajikan sebagai *fixed-point* memungkinkan penggunaan *memoizing fixed-point* kombinator. Pendekatan kombinator ke fungsi rekursif memungkinkan *caching* ke pemanggilan internal fungsi rekursif dapat dilakukan secara

otomatis. Meskipun metode ini secara signifikan dapat mengurangi waktu eksekusi, ada dua catatan penting yang layak untuk diperhatikan. Pertama, dalam prakteknya kombinator-Y hanya bermanfaat pada bahasa-bahasa fungsional yaitu bahasa yang menerapkan strategi *call-by-name* karena (Y g) divergen untuk sembarang nilai *g* pada bahasa-bahasa imperatif yang berorientasi pada *call-by-value*. Kedua, memahami sistem bilangan biner basis dua tidaklah mudah bagi kita yang sudah biasa menggunakan sistem bilangan desimal basis sepuluh. Demikian juga memahami λ-kalkulus dalam waktu singkat tentulah tidak mudah bagi kita yang sudah terbiasa menggunakan aljabar kalkulus. Karena itu muncul pertanyaan menarik: mungkinkah kita bisa melakukan *memozing* fungsi rekursif tanpa menggunakan pendekatan λ- kalkulus?

DAFTAR PUSTAKA

Hoffman, B. (1992). "Term Rewriting with Sharing and Memoïzation," *Algebraic and Logic Programming: Third International Conference, Proceedings*, H. Kirchner and G. Levi (eds.), 128–142, Italy.

Michie, D. (1968). Memo Functions and Machine Learning. *Nature*, 218, 19–22.

Nash, Trey. (2007) *Accelerated C# 2008*. Berkeley: Apress.

Shoenfield, J. R. (2004). *Recursion Theory* (2nd edition). Massachussets: A K Peters.

Stefanov, S. (2010). *JavaScript Patterns*. Stamford: O'Reilly.