VERIFIKASI ALGORITMA PENGHAPUSAN POHON MERAH-HITAM MIGHT MENGGUNAKAN LIQUID HASKELL

Laporan Tugas Akhir I

Disusun sebagai syarat kelulusan mata kuliah
IF4091/Tugas Akhir I dan Seminar

Oleh
Hafizh Afkar Makmur
NIM : 13514062

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO & INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
Januari 2021

VERIFIKASI ALGORITMA PENGHAPUSAN POHON MERAH-HITAM MIGHT MENGGUNAKAN LIQUID HASKELL

Laporan Tugas Akhir I

Oleh

Hafizh Afkar Makmur

NIM : 13514062

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

Bandung, 20 Januari 2021

Mengetahui,

Pembimbing I,	Pembimbing II,
Riza Satria Perdana, ST., MT.	Yanti Rusmawati, Ph.D.
NIP. 197006091995121002	NIP. 119110077

DAFTAR ISI

DAFTAR LAMPIRAN

DAFTAR GAMBAR

DAFTAR TABEL

BAB I

PENDAHULUAN

I.1 Latar Belakang

Seorang pemrogram menginginkan programnya bebas dari kesalahan. Banyak waktu yang sudah terhabiskan oleh berbagai perusahaan teknologi untuk memastikan bahwa produk yang mereka rilis tidak memiliki kecacatan berarti yang mungkin saja bisa menelan korban jiwa. Kesalahan atau *bug* yang bisa ditangkap di awal proses pengembangan sebuah program bisa diatasi dengan lebih mudah daripada *bug* yang hanya disadari jauh dalam sebuah proses pengembangan. Salah satu metode yang dikembangkan oleh para ilmuwan komputasi untuk bisa menangkap sebanyak-banyaknya atau bahkan seluruh kesalahan yang mungkin terjadi dalam sebuah program pada tahap yang secepat mungkin adalah Metode Formal.

Metode Formal merupakan sebuah teknik untuk memanfaatkan sistem pembuktian matematika untuk membuktikan bahwa sebuah program akan berjalan sebagai mana spesifikasi yang sudah ditetapkan. Teknik ini biasanya dilakukan dengan merepresentasikan bagian-bagian dari spesifikasi kebutuhan yang diinginkan dan program yang dikembangkan ke dalam simbol-simbol matematika yang kemudian akan dibuktikan kecocokannya dengan menggunakan berbagai teknik matematika yang sudah ada. Metode Formal bisa memastikan bahwa seluruh domain dalam fungsi yang diverifikasi sudah memenuhi spesifikasi yang ditetapkan dengan menggunakan pembuktian matematika. Hal ini merupakan keunggulan Metode Formal terhadap metode lain yang harus melakukan verifikasi satu-persatu seluruh anggota dari domain dalam satu fungsi untuk memastikan bahwa fungsi tersebut benar-benar berfungsi sesuai spesifikasi. Hal itu merupakan sesuatu yang membutuhkan usaha yang sangat banyak atau bahkan tak terbatas jika fungsi memiliki kemungkinan masukan yang tak terhingga. Faktanya, Metode Formal

sudah digunakan dan direkomendasikan oleh baik IEC, ESA, FAA, maupun NASA sebagai sebuah subjek yang harus dikuasai oleh setiap ilmuwan maupun teknisi komputasi berikut dengan kompetensi matematika yang dibutuhkan untuk memahaminya (Baier, 2008).

Namun, Metode Formal bukan merupakan sebuah metode yang mudah untuk dipelajari oleh seorang pemrogram yang tidak terlalu tertarik melakukan studi matematika apalagi praktisi pemrograman amatir yang semakin banyak bermunculan di masa ini. Dibutuhkan usaha tambahan untuk menganalisis, mengidentifikasi, dan mengubah karakteristik utama spesifikasi kebutuhan dan program menjadi representasi matematika yang bisa digunakan untuk analisis Metode Formal. Usaha yang dibutuhkan bisa menjadi sangat besar sehingga keuntungan yang bisa diraih dengan menggunakan metode ini dianggap tidaklah lebih baik daripada biaya yang harus dibayar sehingga metode ini hanya digunakan untuk sistem kritis yang memiliki potensi kerugian yang sangat besar untuk setiap kesalahan yang terjadi sehingga beban untuk melakukan Metode Formal bisa dijustifikasi (Pena, 2016). Terlihatlah bahwa jika seseorang menginginkan metode ini untuk lebih banyak digunakan oleh kalangan pemrogram, maka dibutuhkan cara untuk mengurangi usaha dan beban yang dibutuhkan untuk menggunakan metode ini.

Beberapa program untuk verifikasi membutuhkan kode untuk diverifikasi dituliskan kembali secara terpisah dalam bahasa pemrograman khusus tertentu. Pada program-program ini, pengguna harus memastikan bahwa kode yang dituliskan untuk program verifikasi merupakan program yang ekuivalen dengan program yang sebenarnya dengan usahanya sendiri. Beberapa program verifikasi lain tidak membutuhkan penulisan ulang kode program dalam bahasa terpisah yang khusus sehingga tidak perlu ada usaha tambahan untuk melakukan hal tersebut. Salah satu perangkat lunak yang dikembangkan untuk tujuan tersebut adalah Liquid Haskell untuk bahasa pemrograman Haskell. Pada dasarnya bahasa pemrograman berparadigma fungsional sudah mengalami keunggulan untuk pengaplikasian Metode Formal karena sifat bahasa tersebut yang transparan dan tidak

menghasilkan “efek samping” saat mengeksekusi suatu fungsi sehingga seorang analis bisa dengan mudah melihat efek dari interaksi antara berbagai fungsi tanpa harus mempertimbangkan bahwa hasil dari interaksi tersebut mungkin berbeda dengan masukan yang sama; sama seperti sebuah fungsi matematika yang konvensional. Karena itu Haskell sebagai sebuah bahasa fungsional murni juga memungkinkan kemudahan pengaplikasian Metode Formal pada program yang dituliskan pada bahasa tersebut. Liquid Haskell memanfaatkan kemudahan ini dengan mengintegrasikan *SMT Solver* seperti Z3 atau CVC4 dan memanfaatkan perangkat tersebut untuk melakukan verifikasi pada program yang dibuat pada Haskell secara otomatis. Hal ini memungkinkan spesifikasi dituliskan langsung pada program. Kemudian dilakukan verifikasi otomatis program terhadap spesifikasi yang sudah tertulis yang membuat proses Metode Formal menjadi lauh lebih mudah dibandingkan dengan verifikasi yang dilakukan secara manual.

Namun kemudian timbul pertanyaan apakah Liquid Haskell benar-benar bisa memverifikasi semua program yang dituliskan dalam bahasa Haskell. Dibutuhkan berbagai percobaan untuk memastikan bahwa Liquid Haskell bisa langsung digunakan untuk mengecek bahwa program yang sudah dituliskan oleh seseorang sudah memenuhi spesifikasi yang ditetapkan.

Salah satu kasus yang sangat umum dijadikan contoh untuk mempelajari metode formal adalah kasus verifikasi struktur data *Red-Black Tree* atau Pohon Merah-Hitam (RBT). Struktur data ini merupakan struktur data yang sederhana dan memiliki spesifikasi yang sangat jelas dan mudah dituliskan. Bahkan, struktur data ini sudah disebutkan dalam artikel jurnal yang dituliskan oleh Nikki Vazou yang membicarakan pengaplikasian Liquid Haskell di dunia nyata (Vazou et al., 2014). Namun, kasus yang disebutkan merupakan sebuah kasus abstrak sebagai contoh teoretis mengenai solusi untuk memverifikasi sebuah program RBT yang ada di dunia nyata. Masih timbul pertanyaan apakah Liquid Haskell mampu memverifikasi program RBT yang tidak sepenuhnya sesuai dengan contoh teoretis yang disebutkan.

Pada 2010, Matthew Might (Might, 2010) membuat sebuah modifikasi terhadap struktur data RBT dengan tujuan untuk membuat sebuah algoritma penghapusan yang lebih elegan daripada algoritma yang sudah dituliskan sebelumnya. Hal ini dilakukan dengan menambahkan 2 buah warna *Double Black* (BB) dan *Negative Black* (NB) untuk memudahkan jalan kerja algoritma. Dalam artikel selanjutnya dia mencoba untuk memverifikasi programnya dengan menggunakan *library* QuickCheck dalam Haskell. Kemudian pada tahun 2014, Stephanie Weirich (Weirich, 2014) menggunakan sistem tipe Haskell yang sangat ketat bernama GADT (*Generalized Algebraic Datatype*) untuk memastikan bahwa algoritma tersebut benar-benar sesuai dengan spesifikasi yang dikodifikasi dalam tipe-tipe masukan dan keluaran dalam setiap fungsi dalam program. Melihat usaha-usaha sebelumnya dalam memverifikasi algoritma ini, algoritma ini bisa menjadi target yang tepat untuk membuktikan apakah Liquid Haskell bisa memverifikasi kode yang sudah ditulis oleh orang lain di dunia nyata.

I.2 Rumusan Masalah

Berdasarkan latar belakang dan fokus masalah tersebut, dirumuskan beberapa masalah yang sebaiknya diselesaikan:

1. Bagaimana melakukan verifikasi algoritma penghapusan Pohon Merah-Hitam oleh Matt Might menggunakan Liquid Haskell?
2. Apakah Liquid Haskell mampu memverifikasi sebuah implementasi algorithma penghapusan Pohon Merah-Hitam oleh Matt Might tanpa perlu adanya modifikasi terhadap implementasi tersebut?

I.3 Tujuan

Penelitian ini bertujuan untuk:

1. Menggunakan Liquid Haskell untuk memverifikasi program penghapusan Red-Black Tree oleh Matt Might

2. Menganalisis kemampuan Liquid Haskell untuk memverifikasi sebuah kode yang berasal dari dunia nyata tanpa modifikasi dan mengidentifikasi kekurangan Liquid Haskell jika hal itu tidak mampu dilakukan.

I.4 Batasan Masalah

Batasan permasalahan dalam penelitian ini adalah sebagai berikut:

1. Spesifikasi dan Verifikasi Liquid Haskell yang diimplementasikan tidak menggunakan fitur *reflection*, *bounded refinements*, maupun *abstract refinements*.
2. Spesifikasi tidak berisi keterangan bahwa program harus berhenti dan tidak akan berjalan terus-menerus tanpa akhir.
3. Verifikasi tidak akan memaksa seluruh fungsi untuk memiliki totalitas (sebuah fungsi harus bisa menangani seluruh argumen tanpa kecuali).

I.5 Metodologi

Metodologi pengerjaan penelitian ini adalah sebagai berikut:

1. Studi Literatur

Pada tahap ini dilakukan penelitian mendalam terhadap algoritma Red-Black Tree dan Liquid Haskell dengan menggunakan berbagai referensi baik daring maupun luring serta melakukan konsultasi kepada ahli dalam bidang yang berkaitan.

2. Penentuan Spesifikasi Program

Ditentukan spesifikasi yang harus bisa dipenuhi oleh program. Spesifikasi dituliskan berbentuk *precondition* dan *postcondition* untuk setiap fungsi yang ada dalam program.

3. Implementasi Liquid Haskell dan Verifikasi Program

Setiap spesifikasi yang sudah ditentukan sebelumnya diimplementasikan dalam bahasa Liquid Haskell. Kemudian program diverifikasi dengan menggunakan spesifikasi program untuk menentukan apakah program benar-benar memenuhi spesifikasi-spesifikasi tersebut.

I.6 Jadwal Pelaksanaan Tugas Akhir

Tahapan Pengerjaaan	Feb	Mar	Apr	Mei	Jun	Jul	Aug	Sep	Okt
Studi Literatur									
Penentuan Spesifikasi Program									
Implementasi Liquid Haskell dan Verifikasi Program									

Tabel I-1. Jadwal Pengerjaan Tugas Akhir

BAB II

STUDI LITERATUR

II.1 *Red-Black Tree*

Struktur data Pohon Merah-Hitam atau *Red-Black Tree* (RBT) adalah sebuah algoritma yang diciptakan oleh Rudolf Bayer pada 1972 (Bayer, 1972) yang bertujuan untuk membuat struktur data *B-Tree* yang selalu memiliki keseimbangan antara cabang kiri dan kanan. Struktur data ini memungkinkan penambahan, penghapusan, ataupun modifikasi data yang selalu efisien dengan kompleksitas algoritma sebesar O(log n) dengan n adalah jumlah noda dalam pohon. Namun struktur data ini merupakan sebuah struktur data *B-Tree* yang merupakan sebuah pohon yang memungkinkan adanya lebih dari satu angka dalam satu noda atau disebut juga pohon 2-3-4 . Struktur data ini membutuhkan berbagai operasi kompleks yang sulit untuk diimplementasikan di kebanyakan bahasa pemrograman. Karena itu, implementasi selanjutnya dari struktur data ini menggunakan struktur data *Binary Search Tree* yang membuat implementasi menjadi lebih sederhana dari sebelumnya (Sedgewick, 2008).

Pada 1999, Chris Okasaki menunjukkan teknik untuk melakukan penambahan noda pada struktur data tersebut secara fungsional murni (Okasaki, 1999). Hal ini memungkinkan struktur data tersebut untuk diimplementasikan menggunakan bahasa pemrograman fungsional murni. Algoritma tersebut merupakan algoritma yang elegan dan sederhana yang menjadi algoritma yang populer untuk digunakan untuk mengimplementasikan struktur data ini (Germane & Might, 2014). Meskipun Okasaki sudah mengimplementasikan penambahan noda pada Red-Black Tree, dia tidak menjelaskan teknik untuk menghapus noda dari struktur data tersebut sehingga dibutuhkan algoritma tambahan yang juga elegan dan sederhana untuk melengkapi implementasi struktur data Red-Black Tree.

Berangkat dari kebutuhan tersebut, Matthew Might berupaya untuk menuliskan algoritma yang elegan untuk melakukan penghapusan noda pada Red-Black Tree.

Untuk melakukan hal tersebut, Might menambahkan warna baru pada struktur data tersebut yaitu *Double Black* dan *Negative Black* untuk membuat implementasi menjadi lebih elegan dari pada percobaan implementasi algoritma sebelumnya. Meskipun algoritma ini lebih lambat daripada algoritma sebelumnya yang dituliskan oleh Kahrs, kesederhanaan dari algoritma yang dituliskan oleh Might diklaim bisa memudahkan penulisan program yang pada akhirnya bisa dimodifikasi untuk meningkatkan kemampuan sesuai kebutuhan (Germane & Might, 2014).

II.1.1 Objek-Objek Teori Graf

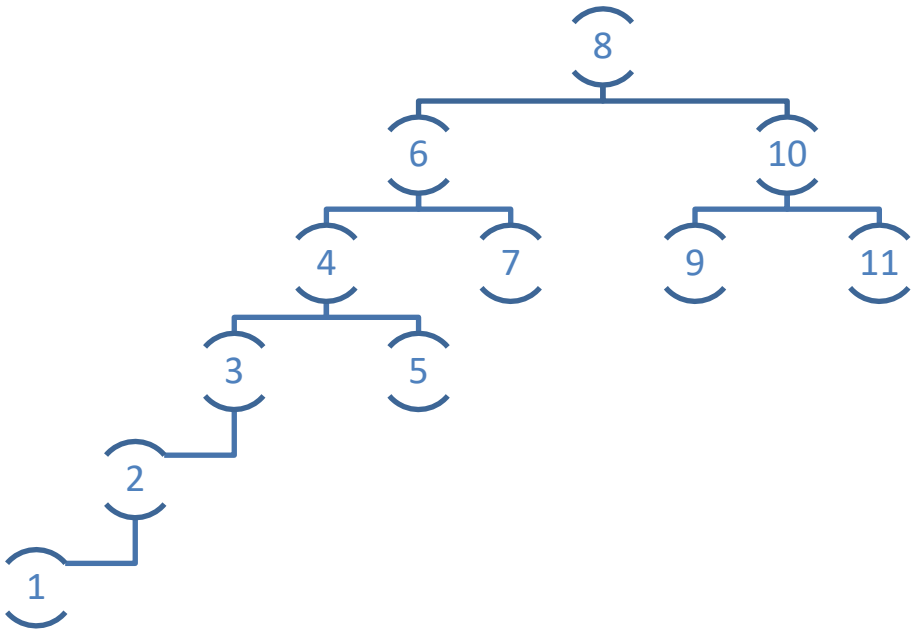
Graf adalah sebuah kumpulan simpul-simpul (*node*) yang dihubungkan oleh sisi (*edge*). Pohon adalah sebuah graf sedemikian sehingga semua simpul hanya dihubungkan oleh maksimal satu lintasan. Graf berarah adalah sebuah graf yang memiliki sisi-sisi yang memiliki arah. Pada pohon, untuk setiap sisi yang berarah, maka simpul yang menjadi sumber dinamakan simpul ayah (*parent node*) dan simpul yang menjadi tujuan dinamakan simpul anak (*child node*). Simpul yang tidak memiliki simpul ayah dinamakan simpul leluhur (*ancestor node*) atau akar (*root*). Simpul yang tidak memiliki simpul anak dinamakan daun (*leaf*). Pohon biner adalah sebuah pohon sedemikian sehingga seluruh simpul hanya maksimal memiliki 1 simpul ayah dan 2 simpul anak.

II.1.2 Pohon Pencarian Biner

Pohon Pencarian Biner atau *Binary Search Tree* (BST) adalah sebuah Pohon Biner yang memiliki untuk setiap simpul maka simpul tersebut memiliki nilai yang lebih besar daripada seluruh nilai pada cabang di sebelah kiri dan lebih kecil dari pada seluruh nilai pada cabang di sebelah kanan. Struktur data ini memiliki sifat bahwa operasi pencarian pada pohon ini bisa dilakukan dengan sangat efisien dengan kompleksitas rata-rata sebesar hanya O(log n).

Kekurangan dari struktur data ini adalah pada struktur data ini mungkin saja salah satu cabang dari pohon jauh lebih panjang dari pada cabang yang lain sehingga program yang mencari sebuah simpul dalam cabang yang panjang itu akan membutuhkan waktu yang lebih lama dari seharusnya. Contohnya, dalam Bagan

II.1, program yang akan mencari simpul “1” akan membutuhkan waktu yang lebih lama daripada program yang akan mencari simpul “11”.



Bagan II.1. Contoh Pohon Pencarian Biner.

Untuk mengatasi masalah ini, cabang-cabang pohon ini harus dijaga agar seimbang sehingga setiap simpul dalam pohon akan bisa dicari dalam waktu yang sama dan konsisten. Ada beberapa algoritma yang mampu melakukan hal ini salah satunya adalah algoritma dalam Pohon Merah-Hitam yang secara otomatis melakukan penyeimbangan setelah setiap operasi.

II.1.3 Pohon Merah-Hitam

Pohon Merah-Hitam atau *Red-Black Tree* (RBT) adalah sebuah Pohon Pencarian Biner khusus yang didesain sehingga setiap cabang dari pohon selalu seimbang. Pada RBT, beberapa simpul memiliki warna merah atau hitam. Seluruh RBT harus mematuhi beberapa properti atau disebut juga *invariant* yaitu:

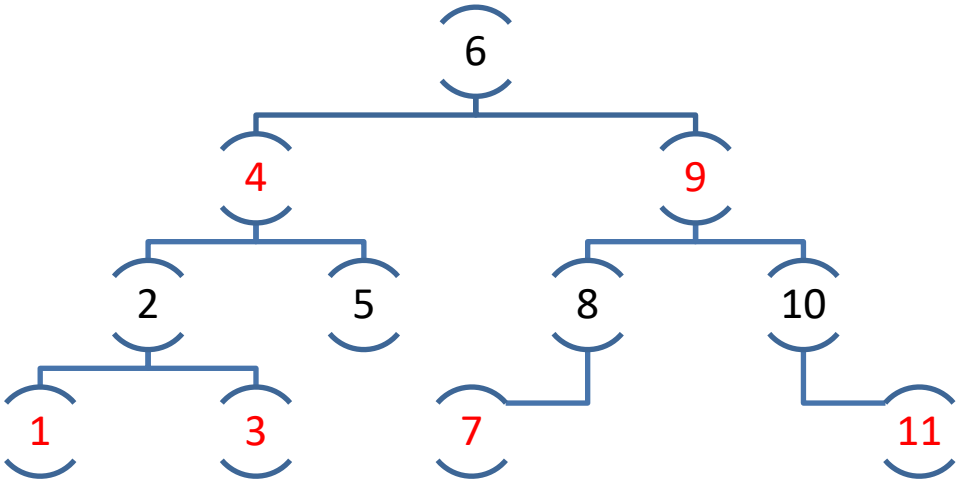
1. Properti BST; seluruh nilai pada simpul kiri dan anak-anaknya harus lebih kecil daripada nilai pada simpul ayah serta seluruh nilai pada simpul kanan dan anak-anaknya harus lebih besar daripada nilai pada simpul ayah.

2. Properti Warna; Simpul anak dari simpul berwarna merah harus memiliki warna hitam.
3. Properti Tinggi; Seluruh jalur dari akar menuju daun harus memiliki jumlah simpul hitam yang sama.

Beberapa literatur menambahkan beberapa properti lain yang harus dipatuhi oleh RBT namun properti-properti ini bisa tidak dipatuhi oleh implementasi tertentu karena tidak mengubah struktur data secara signifikan.

4. Akar harus memiliki warna hitam. Seluruh akar berwarna merah pada RBT yang sudah mematuhi properti sebelumnya bisa diubah menjadi warna hitam tanpa melanggar properti lain sehingga properti ini tidak mengubah apa-apa
5. Seluruh daun harus memiliki warna hitam. Properti ini bisa dipenuhi dengan mengubah RBT yang sudah ada dengan menambahkan dua simpul berwarna hitam pada semua daun pada RBT sehingga properti ini bisa langsung dipenuhi oleh RBT yang sudah memiliki properti-properti sebelumnya.

RBT yang mematuhi seluruh properti tersebut akan selalu memiliki keseimbangan sehingga seluruh operasi pencarian pasti memiliki kompleksitas algoritma maksimal sebesar O(log n) karena cabang dengan tinggi terpanjang yang mungkin, dengan warna merah dan hitam yang berselang-seling, hanya maksimal memiliki tinggi dua kali lipat cabang yang banyak memiliki simpul berwarna hitam (Okasaki, 1999). Sebagai contoh, Bagan II.2 menunjukkan Bagan II.1 yang sudah diubah untuk mematuhi properti-properti RBT. Pada pohon ini, program yang sedang mencari simpul “1” akan membutuhkan waktu yang sama dengan program yang sedang mencari simpul “11”. Seluruh jalur dari akar ke daun pada pohon ini memiliki jumlah simpul berwarna hitam yang sama yaitu sebesar 2.



Bagan II.2. Bagan II.1 yang diubah menjadi RBT.

II.1.4 Algoritma penambahan Okasaki

Algoritma penambahan pada BST biasa sangat mudah. Penambahan bisa dilakukan hanya dengan mencari daun yang memiliki nilai yang paling dekat dengan nilai yang akan dimasukkan kemudian tambahkan nilai tersebut sebagai simpul anak dari simpul tersebut di kiri atau di kanan tergantung apakah nilai yang akan dimasukkan lebih besar atau lebih kecil dari nilai tersebut. Algoritma penambahan untuk BST biasa terlihat seperti ini:

```
insert :: Ord elt => elt -> Set elt -> Set elt
insert E = T E x E
insert (T a y b)
    | x <  y = T (insert a) y b
    | x == y = T a y b
    | x >  y = T a y (insert b)
```

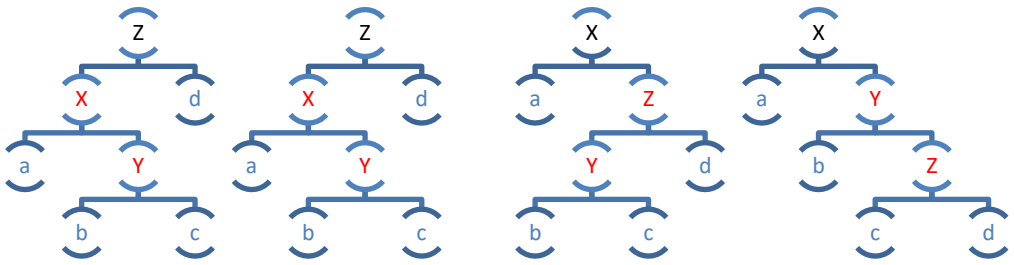
Algoritma penambahan RBT Okasaki tidak jauh berbeda dengan algoritma penambahan BST. Pada algoritma tersebut ditambahkan modifikasi untuk memperhatikan warna dari simpul pada RBT serta sebuat fungsi penyeimbang (*balance*) untuk menjaga properti warna RBT. Algoritma penambahan Okasaki terlihat seperti ini:

```
insert :: Ord elt => elt -> Set elt -> Set elt
insert x s = makeBlack (ins s)
    where
        ins E = T R E x E
        ins (T color a y b)
            | x < y  = balance color (ins a) y b
            | x == y = T color a y b
            | x > y  = balance color a y (ins b)

      makeBlack (T _ a y b) = T B a y b
```

Modifikasi dari fungsi penambahan BST salah satunya adalah penambahan fungsi `makeBlack` untuk membuat simpul akar selalu berwarna hitam. Juga dipastikan bahwa simpul daun yang akan ditambahkan akan memiliki warna merah. Hal ini dilakukan untuk memastikan bahwa properti tinggi pasti akan tetap terpenuhi karena simpul merah tidak akan mengubah jumlah simpul hitam pada jalur mana pun. Kemudian untuk menjaga properti warna, ditambahkan fungsi `balance` untuk melihat cabang yang berkemungkinan melanggar properti warna dan kemudian menyeimbangkannya.

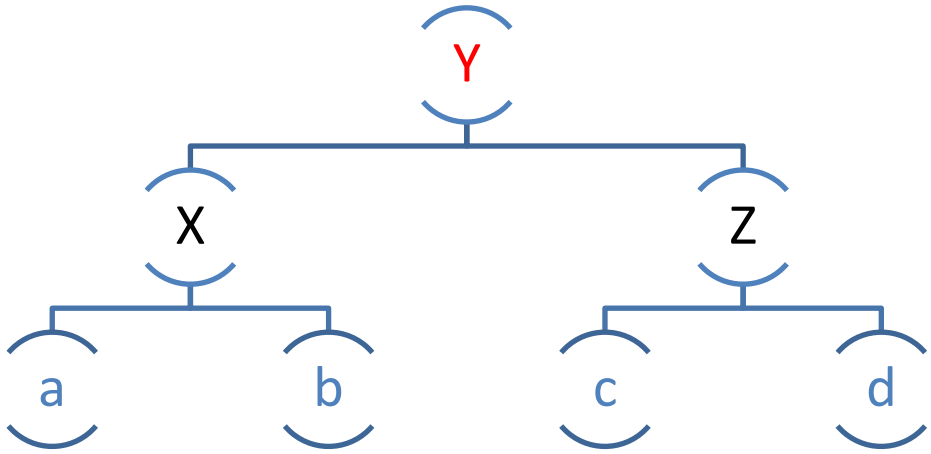
Ada 4 kasus yang membuat pohon hasil penambahan melanggar properti warna. Keempat kasus itu bisa dilihat pada Bagan II.3. Kasus-kasus ini mungkin terjadi ketika sebuah simpul daun berwarna merah ditambahkan sebagai simpul anak pada sebuah simpul berwarna merah.



Bagan II.3. Empat kasus pelanggaran properti warna

Algoritma Okasaki mengubah seluruh pohon ini menjadi sebuah pohon yang berbentuk seperti dapat dilihat pada Bagan II.4. Pohon ini akan memenuhi seluruh

properti RBT baik properti BST, properti tinggi, maupun properti warna (Okasaki, 1999).



Bagan II.4. Pohon hasil penyeimbangan

Pada bahasa pemrograman fungsional, fungsi penyeimbang ini bisa ditulis dengan sangat mudah dalam bentuk sebagai berikut:

```
balance B(T R (T R a x b) y c)z d = T R (T B a x b) y (T B c z d)
balance B(T R a x (T R b y c))z d = T R (T B a x b) y (T B c z d)
balance B a x (T R(T R b y c)z d) = T R (T B a x b) y (T B c z d)
balance B a x(T R b y(T R c z d)) = T R (T B a x b) y (T B c z d)
balance color a x b = T color a x b
```

Algoritma ini akan menghasilkan simpul berwarna merah pada simpul paling atas yang bisa saja memiliki simpul ayah berwarna merah. Untuk itu jika simpul tersebut memiliki simpul kakek (simpul ayah dari simpul ayah) berwarna hitam maka algoritma ini bisa kembali dijalankan terhadap simpul kakek tersebut untuk menyeimbangkannya. Jika simpul ayah tersebut merupakan simpul akar, maka simpul tersebut bisa diberi warna hitam dan kemudian seluruh pohon akan menjadi RBT yang valid.

II.1.5 Algoritma penghapusan Matt Might

Meskipun Okasaki sudah membuat algoritma penambahan yang elegan, dia tidak membuat algoritma penghapusan untuk melengkapinya. Ada beberapa upaya untuk melengkapi algoritma penghapusan tersebut salah satunya adalah algoritma yang dibuat oleh Kahrs (Kahrs, 2001) namun Matthew Might menganggap bahwa kode tersebut terlalu kompleks sehingga kode tersebut sulit untuk diimplementasikan pada bahasa selain Haskell (Might, 2010). Oleh karena itu, Might berupaya untuk membuat sebuah algoritma penghapusan yang sama elegannya dengan algoritma penambahan yang dibuat oleh Okasaki.

Salah satu alasan mudahnya pembuatan algoritma penambahan RBT adalah simpul yang ditambahkan akan selalu menjadi simpul daun sehingga keutuhan properti bisa dengan mudah dijaga. Namun, pada saat penghapusan, bisa saja terjadi penghapusan dilakukan terhadap simpul yang berada jauh di dalam pohon sehingga algoritma yang akan melakukan penghapusan harus memikirkan cara untuk memodifikasi pohon setelah penghapusan namun tetap menjaga seluruh properti RBT pada pohon.

Prosedur penghapusan sebuah simpul akan berbeda tergantung dengan jumlah simpul anak yang simpul itu miliki. Jika simpul itu memiliki dua simpul anak, maka penghapusan dilakukan dengan menghapus salah satu simpul dari cabang sebelah kiri yang memiliki nilai paling tinggi, kemudian mengganti nilai dari simpul target dengan nilai simpul yang baru saja dihapus. Jika simpul tersebut memiliki satu simpul anak, hanya ada satu simpul yang mungkin memiliki satu simpul anak, yaitu sebuah simpul berwarna hitam yang memiliki simpul berwarna merah. Untuk kasus itu, maka prosedur penghapusan adalah kembali untuk menghapus simpul anak tersebut dan kemudian mengganti nilai simpul target menjadi nilai simpul yang baru saja dihapus. Untuk sebuah simpul daun berwarna merah, simpul bisa langsung dihapus tanpa mengganggu properti apa pun.

Satu kasus yang akan berpotensi merusak properti tinggi adalah jika simpul yang akan dihapus adalah sebuah simpul daun berwarna hitam. Jika simpul ini dihapus, maka salah satu jalur dari akar ke daun akan memiliki jumlah simpul hitam yang

berkurang satu sehingga seluruh jalur lain harus dimodifikasi untuk mengakomodasi penghapusan tersebut. Hal ini mungkin akan menjadi sebuah operasi yang sulit karena keseluruhan pohon harus diubah untuk kembali mematuhi properti tinggi tersebut.

Might menyelesaikan masalah ini dengan menambahkan dua warna pada struktur data RBT yaitu *Double Black* (BB) dan *Negative Black* (NB). Simpul dengan warna BB akan dihitung sebagai 2 simpul hitam untuk kalkulasi properti tinggi. Sebaliknya, simpul dengan warna NB akan dihitung sebagai -1 untuk kalkulasi tersebut. Untuk menyelesaikan masalah penghapusan simpul daun hitam, maka setelah penghapusan tersebut maka warna dari simpul ayah dari simpul tersebut akan ditambahkan warna hitam. Jadi, jika simpul tersebut sebelumnya berwarna merah, maka simpul tersebut akan menjadi berwarna hitam. Jika sebelumnya simpul tersebut memiliki warna hitam, maka simpul tersebut akan menjadi berwarna BB. Dengan demikian, properti tinggi akan kembali terpelihara setelah terjadi penghapusan dalam RBT ini. Kode untuk prosedur penghapusan ini akan berbentuk seperti ini:

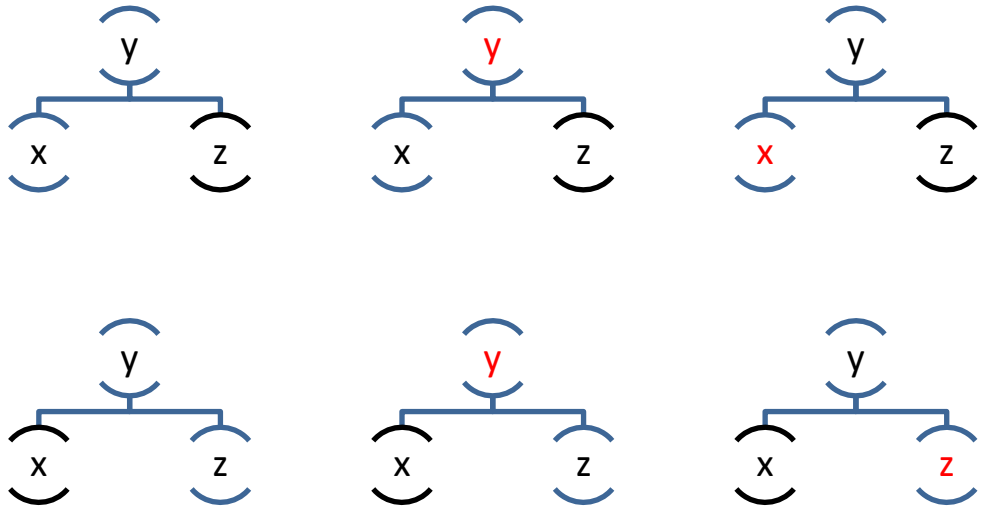
```
remove :: RBSet a -> RBSet a
-- ; Penghapusan daun
remove (T R E _ E) = E
remove (T B E _ E) = EE
-- ; Penghapusan simpul dengan satu anak
remove (T B E _ (T R a x b)) = T B a x b
remove (T B (T R a x b) _ E) = T B a x b
-- ; Penghapusan simpul dengan dua anak
remove (T color l y r) = bubble color ll mx r
   where mx = max l
         ll = removeMax l

removeMax :: RBSet a -> RBSet a
removeMax s@(T _ _ _ E) = remove s
removeMax s@(T color l x r) = bubble color l x (removeMax r)
```

Fungsi `max` adalah fungsi yang mendapatkan nilai maksimum yang berada pada suatu pohon. Fungsi `removeMax` adalah fungsi yang melakukan prosedur penghapusan kepada simpul yang memiliki nilai paling besar pada suatu pohon.

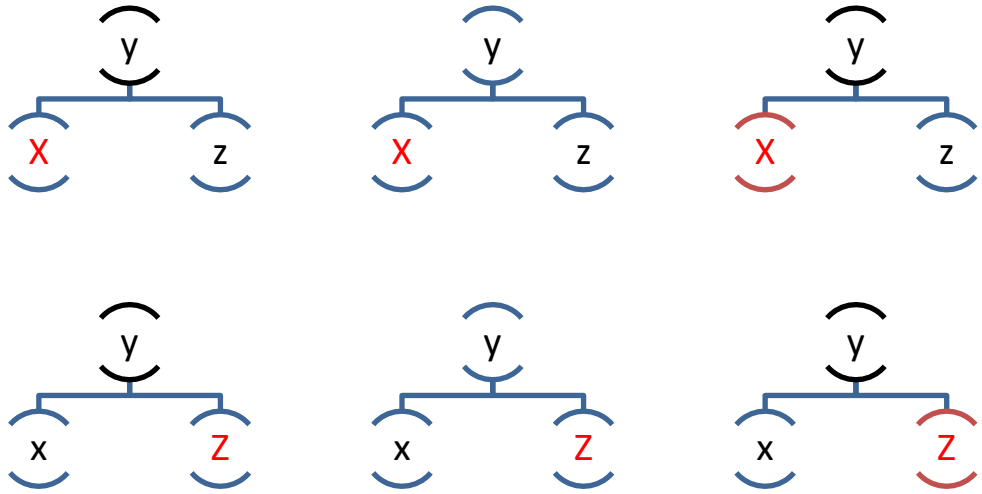
Operasi ini akan meninggalkan sebuah simpul berwarna bukan merah dan bukan hitam dalam pohon. Karena itu, maka pohon ini masih harus dimodifikasi untuk

menghilangkan warna BB tersebut. Langkah pertama untuk menghilangkan warna ini adalah dengan melakukan operasi bernama *Bubble* (gelembung) yang berusaha mengangkat warna BB tersebut naik sampai ke simpul akar atau menghilangkan warna itu sama sekali jika memungkinkan. Ada 6 pohon yang mungkin memiliki simpul berwarna BB seperti dapat dilihat dalam Bagan II.5.



Bagan II.5. Enam pohon yang memiliki simpul berwarna BB

Fungsi *bubble* akan mengangkat atau menghilangkan warna BB pada pohon tersebut sehingga menjadi seperti terlihat pada Bagan II.6.



Bagan II.6. Hasil operasi fungsi *bubble*

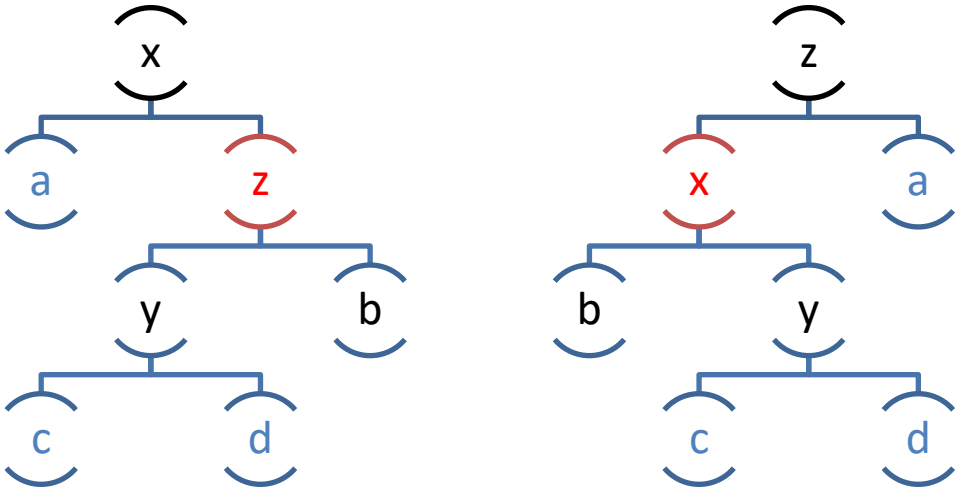
Simpul X dan Z menandakan simpul yang mungkin memiliki pelanggaran properti warna sehingga harus dijalankan fungsi penyeimbang khusus untuk simpul-simpul tersebut. Pada dasarnya, fungsi *bubble* hanya mengurangi warna hitam dari seluruh simpul anak dan kemudian menambah warna hitam pada simpul ayah. Karena itu, fungsi ini memiliki kode yang sangat pendek.

```
bubble :: Color -> a -> RBSet a -> RBSet a -> RBSet a
bubble color x l r
 | isBB(l) || isBB(r) = balance (blacker color) x (redder' l)
(redder' r)
 | otherwise          = balance color x l r
```

Fungsi `isBB` adalah fungsi yang mendeteksi apakah suatu simpul memiliki warna BB atau tidak. Jika tidak ada simpul anak yang memiliki warna BB, maka fungsi *bubble* bisa langsung dilewati dan program akan melakukan prosedur penyeimbangan.

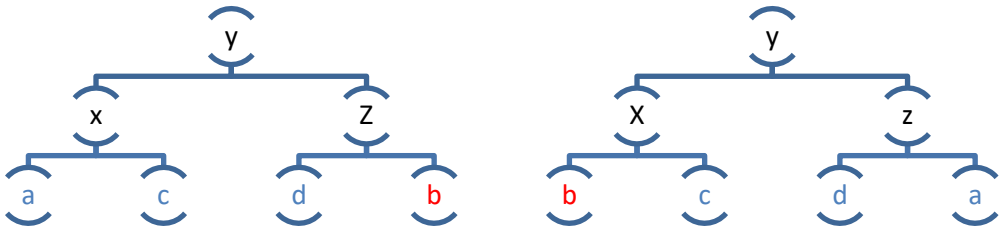
Untuk proses penyeimbangan, seluruh simpul berwarna merah dan hitam bisa diseimbangkan dengan algoritma yang sama seperti yang telah ditulis oleh Okasaki terutama seperti pada kasus-kasus yang ditunjukkan pada Bagan II.3. Untuk pohon-pohon dengan warna BB dan NB, harus ditambahkan beberapa kode untuk menangani kasus tersebut. Salah satu kasus adalah kasus yang sama persis seperti yang ditunjukkan pada Bagan II.3, namun dengan simpul akar berwarna BB. Untuk

kasus tersebut, maka pohon tersebut akan diseimbangkan sehingga menjadi pohon-pohon pada Bagan II.4, namun dengan simpul akar berwarna hitam dan bukan merah. Kemudian ada satu kasus khusus yang memiliki pohon berwarna BB dan NB sekaligus sebagaimana ditunjukkan pada Bagan II.7 dengan simpul z berwarna BB, simpul x berwarna NB, simpul b, w, dan y berwarna hitam, dan simpul a, c, dan d berwarna merah atau hitam.



Bagan II.7. Pohon dengan simpul berwarna BB dan NB

Pohon-pohon ini akan diseimbangkan menjadi dua pohon yang serupa seperti dapat dilihat pada Bagan II.8. Pohon ini masih memiliki potensi pelanggaran properti warna pada simpul X dan Z namun hal itu bisa diselesaikan dengan melakukan algoritma penyeimbang hanya sekali pada simpul X dan Z tersebut dan kemudian pohon ini akan menjadi RBT yang valid (Might, 2010) karena prosedur penyeimbangan yang dilakukan terhadap pohon yang tidak memiliki warna BB dan NB akan merupakan prosedur yang sama dengan prosedur penyeimbangan Okasaki yang tidak akan memanggil fungsi apa-apa lagi.



Bagan II.8. Pohon hasil penyeimbangan

Fungsi penyeimbang yang sudah ditambahkan kode-kode untuk menangani warna BB dan NB terlihat seperti ini:

```
balance :: Color -> RBSet a -> a -> RBSet a -> RBSet a

 -- Kasus-kasus Okasaki:
balance B(T R (T R a x b) y c) z d = T R (T B a x b) y (T B c z d)
balance B(T R a x (T R b y c)) z d = T R (T B a x b) y (T B c z d)
balance B a x(T R (T R b y c) z d) = T R (T B a x b) y (T B c z d)
balance B a x(T R b y (T R c z d)) = T R (T B a x b) y (T B c z d)

 -- 6 kasus penghapusan Might:
balance BB(T R(T R a x b) y c) z d = T B (T B a x b) y (T B c z d)
balance BB(T R a x(T R b y c)) z d = T B (T B a x b) y (T B c z d)
balance BB a x (T R(T R b y c)z d) = T B (T B a x b) y (T B c z d)
balance BB a x(T R b y(T R c z d)) = T B (T B a x b) y (T B c z d)

balance BB a x (T NB (T B b y c) z d@(T B _ _ _))
    = T B (T B a x b) y (balance B c z (redden d))
balance BB (T NB a@(T B _ _ _) x (T B b y c)) z d
    = T B (balance B (redden a) x b) y (T B c z d)

balance color a x b = T color a x b
```

Fungsi `redden` adalah fungsi yang mengubah simpul berwarna hitam menjadi simpul berwarna merah. Hal ini dilakukan untuk menjaga jumlah simpul hitam untuk perhitungan properti tinggi tanpa perlu mengubah warna dari simpul lain

Kemudian pada akhirnya kode dari fungsi penghapusan tidak jauh berbeda dengan kode dari fungsi penambahan.

```
delete :: (Ord a) => a -> RBSet a -> RBSet a
delete x s = makeBlack (del x s)
    where
        del x E = E
        del x s@(T color aa y bb)
```

```
            | x < y     = bubble color (del x aa) y bb
            | x > y     = bubble color aa y (del x bb)
            | otherwise = remove s

        makeBlack (T _ a y b) = T B a y b
        makeBlack EE = E
```

II.2 Metode Formal

II.2.1 Definisi Metode Formal

Metode Formal merupakan sebuah teknik penerapan prinsip-prinsip matematika dalam memodelkan dan menganalisis sistem ICT (Teknologi Komunikasi dan Informasi) (Baier & Katoen, 2008). Dengan kata lain, Metode Formal didefinisikan sebagai ilmu matematika untuk sistem perangkat keras dan perangkat lunak komputer (Holloway, 1997). Menurut Monin, sebenarnya istilah "Teknik Formal" lebih pantas untuk digunakan dibandingkan "Metode Formal" karena teknik ini belum memiliki metodologi yang baku. Namun karena istilah Metode Formal lebih populer maka untuk selanjutnya dalam tulisan ini istilah Metode Formal akan menjadi istilah yang digunakan (Monin, 2003).

Metode Formal merupakan salah satu teknik verifikasi yang "sangat direkomendasikan" untuk pengembangan perangkat lunak dan sistem kritis oleh standar praktik terbaik yang dimiliki oleh IEC (Komisi Elektroteknik Internasional) dan ESA (Agensi Antariksa Eropa) (Baier & Katoen, 2008). Hasil laporan investigasi yang dilakukan oleh FAA (Otoritas Penerbangan Federal Amerika) dan NASA (Administrasi Aeronautika dan Antariksa Nasional Amerika) mengenai penggunaan Metode Formal menunjukkan bahwa Metode Formal haruslah menjadi bagian dari pendidikan seluruh ilmuwan komputasi dan insinyur sistem perangkat lunak, seperti bagaimana ilmu matematika terapan merupakan ilmu yang wajib dimiliki oleh setiap insinyur lainnya.

II.2.1.1 Alasan penggunaan Metode Formal

II.2.1.1.1 Fatalitas kesalahan sistem perangkat lunak

Kesalahan pada unit pembagian bilangan titik mengambang (*floating point*) milik Intel pada tahun 90an menyebabkan kerugian mencapai 470 juta dolar Amerika untuk mengganti seluruh prosesor yang cacat dan juga merusak reputasi Intel sebagai pembuat cip komputer yang bisa diandalkan. Kesalahan perangkat lunak dalam sistem penanganan bagasi memundurkan pembukaan sebuah bandara di Denver selama 9 bulan yang menyebabkan kerugian sekitar 1.1 juta Dolar Amerika per hari. Kesalahan selama 24 jam pada sebuah sistem pemesanan tiket daring internasional akan menyebabkan kebangkrutan perusahaan tersebut karena hilangnya pesanan. Bila kesalahan terjadi pada sistem keamanan, akibat yang ditimbulkan bisa menjadi bencana. Kecacatan fatal pada perangkat lunak kontrol pada misil Ariadne-5, wahana antariksa Mars Pathfinder, serta pesawat-pesawat milik Airbus sudah menjadi berita utama di seluruh dunia dan menjadi kasus-kasus yang terkenal. Perangkat lunak juga digunakan pada kontrol proses sistem yang kritis pada pabrik kimia, pembangkit listrik tenaga nuklir, sistem lalu lintas, sistem penghalau badai, serta sistem penting lainnya yang bisa menimbulkan bencana dan kerugian besar jika terjadi kesalahan pada sistem tersebut. Salah satu contohnya adalah sebuah kecacatan perangkat lunak pada mesin terapi radiasi Therac-25 yang menyebabkan kematian 6 pasien kanker di antara 1985 dan 1987 disebabkan oleh overdosis paparan radiasi. Meningkatnya ketergantungan aplikasi kritis pada pemrosesan informasi menyebabkan pentingnya meningkatkan reliabilitas dalam proses desain sistem ICT.

II.2.1.1.2 Reliabilitas pengembangan perangkat lunak

Pengembangan perangkat lunak saat ini sudah terkenal sebagai sebuah proses yang lambat dalam memberikan hasil serta sulit diprediksi dan tidak bisa diandalkan dalam operasi (Holloway, 1997). Menurut sebuah artikel yang ditulis pada 1994 oleh Wyatt Gibbs, “Studi menunjukkan bahwa untuk setiap 6 sistem perangkat lunak skala besar baru yang dioperasikan, 2 akan dibatalkan. Rata-rata waktu pengembangan perangkat lunak melampaui jadwal yang ditentukan sebanyak 50%. Perangkat lunak yang lebih besar bahkan membutuhkan waktu yang lebih lama lagi. 75% dari seluruh sistem skala besar memiliki kesalahan dalam operasi yang

menyebabkan mereka tidak berfungsi sebagaimana mestinya atau bahkan tidak digunakan sama sekali.” Dibandingkan dengan disiplin *engineering* lainnya, teknik perangkat lunak terlihat sangat buruk. Namun ini tidak terlalu mengejutkan karena setidaknya dalam dua aspek, perangkat lunak berbeda dengan objek fisik, material, dan sistem yang ditangani oleh ilmu teknik pada umumnya.

Pertama, pada sistem fisik perubahan yang halus pada masukan akan menghasilkan perubahan yang halus pada keluaran. Dengan kata lain, sistem fisik merupakan sebuah sistem yang kontinu. Hal ini memungkinkan perilaku sistem untuk ditentukan hanya dengan memberikan beberapa masukan yang kemudian diterjemahkan menggunakan interpolasi dan ekstrapolasi untuk menentukan perilaku sistem pada masukan yang tidak dites. Sistem perangkat lunak, berbeda dengan sistem fisik, merupakan sebuah sistem yang diskontinu. Perubahan kecil pada masukan bisa menghasilkan perubahan yang signifikan pada beberapa penentuan keputusan dalam perangkat lunak dan menyebabkan keluaran yang sangat berbeda. Hasilnya, interpolasi atau ekstrapolasi tidak bisa digunakan untuk memprediksi keluaran dari masukan yang tidak dites karena risiko yang tinggi dan bisa menyebabkan hasil yang tidak diinginkan. Selain itu sistem komputer semakin hari memiliki kompleksitas yang semakin tinggi. Dengan naiknya kompleksitas maka semakin banyak pula kemungkinan kecacatan desain yang akan terjadi. Pada sisi baiknya, sistem perangkat lunak cenderung tidak memiliki keausan tidak seperti sistem fisik. Sehingga jika sistem perangkat lunak sudah dibuktikan reliabilitasnya maka reliabilitas itu bisa bertahan selama bertahun-tahun tanpa mengalami keusangan.

II.2.1.2 Contoh pengaplikasian Metode Formal

Salah satu eksperimen skala besar pengaplikasian Metode Formal dilakukan pada proyek CICS yang dilakukan oleh IBM (Taman Huxley, Inggris) dalam kerja sama dengan Universitas Oxford (Baier & Katoen, 2008). Tujuan dari proyek ini adalah untuk melakukan restrukturisasi mayor terhadap sebuah sistem manajemen transaksi besar yang sudah berjalan. Dalam keseluruhan sistem terdapat 800.000 baris kode yang tertuliskan dalam bahasa Assembly dan Pias, sebuah bahasa tingkat

tinggi khusus. Ada 268.000 baris kode yang dimodifikasi dan ditulis ulang dan di dalam baris-baris kode itu 37.000 baris dalam kode tersebut diberikan spesifikasi formal menggunakan notasi Z. Berbagai prosedur pengukuran dilakukan untuk mengevaluasi dampak Metode Formal terhadap produktivitas dan kualitas. Hasilnya adalah sebagai berikut:

1. Biaya pengembangan berkurang 9 persen
2. Terjadi 2.5 kali lebih sedikit *error* pada bagian program yang dikembangkan menggunakan notasi Z dalam 8 bulan pertama setelah instalasi
3. *Error* yang dilaporkan memiliki keseriusan lebih rendah

Eksperimen ini merupakan sebuah eksperimen yang menarik karena banyaknya kode yang terlibat. Namun, eksperimen ini memiliki cakupan yang terbatas karena eksperimen hanya melakukan eksperimen terhadap notasi formal Z serta tidak memperhitungkan teknik-teknik pembuktian yang dilakukan.

II.2.1.3 Kelemahan Metode Formal

Metode Formal bukan merupakan sebuah teknik adiguna yang bisa digunakan untuk memverifikasi seluruh program yang dibuat. Menurut Jean-Francois Monin (Monin, 2003), beberapa kelemahan dari teknik ini yang harus diperhatikan adalah:

1. Selalu ada jarak antara spesifikasi format yang tertulis dengan objek yang direpresentasikan. Hal ini serupa seperti yang terjadi pada ilmu fisika. Tidak bisa dibuktikan bahwa hukum-hukum fisika benar-benar merepresentasikan dunia nyata namun bisa diyakini bahwa hukum-hukum itu cukup dekat dengan dunia nyata untuk tujuan saat ini.
2. Dibutuhkan waktu untuk memahami dan menggunakan notasi yang dipakai. Keyakinan terhadap kebenaran dari sebuah spesifikasi program hanya bisa diketahui dengan proses analisis yang mendalam. Metode Formal juga membutuhkan aspek teori serta kemampuan untuk memanipulasi berbagai teknik matematika yang ada.
3. Lebih banyak waktu yang akan dihabiskan pada tahap awal program (spesifikasi, desain). Eksperimen menunjukkan bahwa waktu yang dihabiskan

pada tahap awal ini biasanya akan di kompensasi dengan waktu yang berkurang pada tahap akhir (tes, integrasi). Proses metode formal membuka di awal beberapa masalah yang biasanya hanya akan ditemukan pada proses akhir seperti saat *debugging* yang mungkin akan membutuhkan penanganan yang lebih besar karena sulitnya mengubah program yang sudah dibuat secara besar-besaran. Berbagai kesulitan yang ditemukan dalam melakukan metode formal sebenarnya merupakan refleksi terhadap kesulitan proyek yang dilakukan namun proses permodelan menunjukkan kompleksitas permasalahan yang tidak terlihat pada pandangan pertama.

II.2.1.3.1 Minimnya penggunaan Metode Formal

Pada kenyataannya setelah 40 tahun Metode Formal dengan segala kelebihan dan kekurangannya masih sangat jauh dari pengaplikasian pada pemrograman sehari-hari. Menurut Ricardo Pena (Peña, 2017), ada beberapa alasan atas terjadinya situasi ini:

1. Dibutuhkan waktu dan usaha yang cukup besar untuk membuat spesifikasi formal pada kebutuhan dengan menuliskan *precondition* dan *postcondition* untuk setiap kebutuhan.
2. Dibutuhkan usaha yang lebih besar lagi untuk menentukan *loop invariant* (variabel yang selalu konstan selama sebuah *loop* berjalan) serta menentukan asumsi lain yang kritis dalam program.
3. Bahkan setelah menuliskan seluruh hal tersebut, untuk menuliskan pembuktian program tersebut secara manual dibutuhkan ruang bahkan mencapai 5 sampai 10 kali panjang program yang dibuktikan.

Pada umumnya, verifikasi formal bisa memberikan manfaat yang jelas namun hal tersebut membutuhkan investasi usaha yang sangat tinggi. Hal ini menyebabkan Metode Formal jarang digunakan dan hanya dilakukan untuk program kritis yang akan memberikan kerugian yang sangat besar untuk setiap kesalahan yang terjadi sehingga investasi yang besar untuk melakukan Metode Formal bisa dijustifikasi. Usaha besar yang dibutuhkan untuk melakukan Metode Formal ini harus dikurangi

untuk bisa memungkinkan Metode Formal untuk digunakan lebih luas lagi kepada lebih banyak pemrogram yang membutuhkannya.

II.2.2 Spesifikasi dan Verifikasi Program

Pada dasarnya, dua komponen paling utama dalam melakukan Metode Formal adalah spesifikasi dan verifikasi. Spesifikasi merupakan penulisan kebutuhan yang dimiliki dalam notasi yang formal. Verifikasi adalah proses untuk membuktikan kebenaran suatu program terhadap spesifikasi yang sudah dituliskan sebelumnya.

II.2.2.1 Spesifikasi Program

Representasi paling sederhana untuk spesifikasi program adalah pasangan (*precondition*, *postcondition*) (Monin, 2003). *Precondition* adalah asumsi mengenai kondisi yang relevan yang terjadi sebelum eksekusi program. *Postcondition* adalah asumsi mengenai hasil yang diinginkan setelah program yang dilakukan. Asumsi-asumsi tersebut dituliskan menggunakan formula logis yang memiliki arti matematis sehingga bisa dilakukan kalkulasi matematika terhadap asumsi tersebut. Verifikasi kemudian merupakan sebuah proses untuk membuktikan bahwa sebuah program yang memenuhi *precondition* harus melakukan aksi yang pada akhirnya memenuhi *postcondition*.

Berbagai properti yang dituliskan dalam spesifikasi bisa cukup elementer seperti menyatakan bahwa hasil tidak pernah melampaui nilai tertentu, program akan selalu berakhir (tidak berjalan selamanya), dan seterusnya. Verifikasi bergantung pada spesifikasi dalam menentukan apa yang program harus dan tidak boleh lakukan. Kesalahan hanya ditemukan jika program tidak memenuhi spesifikasi tertentu. Sistem dianggap “benar” jika sistem mampu memenuhi seluruh spesifikasi. Jadi kebenaran suatu program selalu merupakan sebuah properti yang relatif terhadap spesifikasi bukan merupakan sebuah properti yang absolut pada sistem.

II.2.2.2 Verifikasi Program

Menurut Michael Huth (Huth & Ryan, 2004), ada berbagai pendekatan dalam melakukan verifikasi:

1. Berbasis Bukti dan Berbasis Model. Dalam Verifikasi Berbasis Bukti, deskripsi sistem dituliskan dalam kumpulan formula Γ dan spesifikasi dituliskan dalam formula yang lain φ. Metode verifikasi kemudian mencoba mencari bukti bahwa Γ |− φ atau dengan kata lain Γ mengimplikasikan φ. Hal ini biasanya membutuhkan panduan dan keahlian pengguna. Dalam pendekatan berbasis model, sistem direpresentasikan sebagai sebuah model M dalam teori logika yang sesuai. Spesifikasi kembali direpresentasikan sebagai formula φ dan metode verifikasi menentukan apakah model M memenuhi φ (ditulis M φ). Komputasi ini biasanya bisa dilakukan secara otomatis untuk model berhingga.
2. Derajat otomatisasi. Verifikasi memiliki berbagai derajat untuk seberapa otomatis metode verifikasi bisa dijalankan mulai dari dilakukan dengan manual secara keseluruhan ataupun otomatis. Banyak teknik yang menggunakan komputer berada di antara dua ekstrem itu.
3. Properti atau keseluruhan. Spesifikasi mungkin hanya mendeskripsikan sebagian properti dari sebuah sistem atau mungkin keseluruhan perilaku. Dibutuhkan usaha yang jauh lebih berat untuk memverifikasi spesifikasi yang mendeskripsikan keseluruhan perilaku sistem.
4. Domain aplikasi. Domain aplikasi bisa memiliki banyak arti mulai dari apakah verifikasi dilakukan pada perangkat lunak atau perangkat keras, berurutan atau paralel, memiliki akhir atau reaktif, dan sebagainya. Pada dasarnya verifikasi pada perangkat keras jauh lebih vital untuk dilakukan secepat mungkin karena biaya untuk mengganti bagian kode yang salah pada sistem perangkat keras jauh lebih tinggi dibandingkan dengan sistem perangkat lunak.
5. Sebelum atau sesudah pengembangan. Verifikasi akan memberikan manfaat yang lebih baik jika dilakukan di awal pengembangan sistem karena *error* yang ditangkap akan bisa ditangani dengan biaya yang lebih murah.

Menurut Nikki Vazou (Vazou et al., 2017), kemungkinan perbedaan pendekatan lain adalah apakah verifikasi dilakukan secara intrinsik atau ekstrinsik. Verifikasi intrinsik dilakukan terhadap program secara langsung sedangkan verifikasi ekstrinsik membutuhkan kode khusus lain yang khusus dituliskan untuk tujuan verifikasi. Contoh dari sistem verifikasi yang memiliki perbedaan ini adalah Liquid

Haskell dan Coq. Liquid Haskell mampu menjalankan verifikasi intrinsik karena spesifikasi dituliskan bersamaan dengan program dan SMT Solver dapat dilakukan untuk menganalisis isi program sekaligus dengan spesifikasi yang tertuliskan secara otomatis. Pada Coq pembuktian harus dilakukan secara manual oleh pengguna sehingga pembuktian harus dituliskan oleh pengguna sendiri dalam bahasa Coq. Karena pembuktian pada Liquid Haskell merupakan pembuktian implisit yang dilakukan oleh SMT Solver sedangkan pembuktian pada Coq dituliskan secara eksplisit oleh pengguna, maka pembuktian yang ditulis pada verifikasi ekstrinsik jauh lebih mudah dibaca dan digunakan. Hal ini juga dikarenakan narasi pembuktian hanya akan tertulis dalam bahasa verifikasi tersebut tidak bercampur dengan kode implementasi program.

II.3 Liquid Haskell

Liquid Haskell adalah sebuah program verifikasi yang bisa digunakan untuk memverifikasi sebuah program Haskell. Program ini memanfaatkan *SMT Solver* seperti Z3 untuk melakukan verifikasi terhadap sebuah program.

II.3.1 Penulisan Spesifikasi

Liquid Haskell melakukan verifikasi dengan mengecek spesifikasi yang ditulis dengan bahasa khusus kemudian membandingkannya dengan fungsi yang berkaitan dengan spesifikasi tersebut. Spesifikasi dituliskan dengan menggunakan bahasa Haskell yang dimodifikasi.

Setiap spesifikasi dituliskan dalam blok komentar `{-@ @-}`. Blok komentar ini tidak akan dibaca oleh *compiler* Haskell sehingga penulisan spesifikasi ini tidak akan mengubah kompilasi program sama sekali. Namun, Liquid Haskell akan membaca blok-blok komentar itu saat proses verifikasi.

Spesifikasi dituliskan seperti *signature type* yang sudah biasa dituliskan untuk setiap fungsi dalam Haskell namun dengan tipe masing-masing yang diubah. Format penulisan tipe tersebut adalah

```
{nama variabel : tipe | kondisi}
```

Nama variabel bisa diisi apa pun dan akan menjadi nama variabel yang akan digunakan dalam penulisan kondisi. Nama variabel tersebut juga bisa direferensi oleh kondisi pada tipe-tipe selanjutnya dalam fungsi yang sama. Tipe merupakan tipe asli dari parameter tersebut. Kondisi adalah pembatasan yang dikenakan pada parameter tersebut. Jika parameter itu adalah masukan, maka kondisi bersifat sebagai *precondition*. Berarti, fungsi tersebut tidak akan menerima parameter tersebut kecuali jika parameter tersebut mematuhi kondisi yang dituliskan. Jika parameter itu adalah keluaran, maka kondisi bersifat sebagai *postcondition*. Kondisi ini menjamin bahwa keluaran program akan mengikuti kondisi yang sudah ditetapkan.

Misalkan ada sebuah fungsi pembalik yang akan mengubah angka negatif menjadi angka positif dan tidak menerima angka 0. Maka fungsi tersebut bersama spesifikasinya akan berbentuk seperti berikut:

```
{-@ inverse :: {x:Int | x > 0} -> {v:Int | v < 0} @-}
inverse :: Int -> Int
inverse x = x * (-1)
```

Spesifikasi pada fungsi ini membatasi bahwa fungsi `inverse` hanya menerima masukan dengan nilai lebih besar dari 0. Spesifikasi tersebut juga menjamin bahwa fungsi ini akan memberikan keluaran sebuah nilai yang lebih kecil dari 0.

Spesifikasi ini juga bisa digunakan dalam memberi batasan pada sebuah struktur data. Misalkan ada struktur data pohon biner berbentuk seperti ini:

```
data Tree a =
   E
 | T a (Tree a) (Tree a)
```

Pemrogram bisa menjamin bahwa nilai-nilai pada cabang pohon kiri lebih kecil daripada nilai pada simpul serta nilai-nilai pada cabang pohon kanan lebih besar daripada nilai pada simpul (properti BST) dengan menuliskan spesifikasi sebagai mana berikut:

```
{-@ data Tree a =
                    E
                  | T { key   :: a
                      , lt    :: Tree {v:a | v < key}
                      , rt    :: Tree {v:a | v > key}
```

```
                        }
@-}
```

Dapat terlihat bahwa spesifikasi menjamin bahwa cabang pohon kiri atau `lt` memiliki nilai yang lebih kecil daripada nilai pada simpul akar atau key dan cabang pohon kanan memiliki atau `rt` memiliki nilai yang lebih besar daripada `key`.

II.3.1.1 Fungsi pembantu

Liquid Haskell mengizinkan penulisan beberapa fungsi pembantu untuk memudahkan penulisan spesifikasi. Ada dua jenis fungsi pembantu yang bisa dituliskan di antaranya adalah *measure* dan *inline*.

Inline merupakan fitur yang berfungsi seperti *alias* pada bahasa C. Fitur ini membantu penulis untuk menuliskan sebuah kondisi yang dibutuhkan berulang-ulang menjadi sebuah bentuk yang jauh lebih singkat dan mungkin lebih deskriptif. Misalkan suatu program membutuhkan pengecekan berulang-ulang bahwa sebuah angka merupakan bilangan genap positif. Dibandingkan menuliskan syarat yang agak panjang tersebut berulang-ulang, bisa dituliskan *inline* sebagaimana berikut:

```
{-@ inline isPositiveEven @-}
isPositiveEven :: Int -> Bool
isPositiveEven x = (x > 0) && (x `mod` 2 == 0)
```

Dengan menggunakan *inline* ini maka jika ada spesifikasi yang membutuhkan syarat ini maka spesifikasi tersebut bisa menuliskan `isPositiveEven x` dan bukan `(x > 0) && (x `mod` 2 == 0)` yang lebih panjang daripada *inline* tersebut.

Measure merupakan fitur untuk mengangkat sebuah fungsi yang pemrogram tulis sehingga menjadi salah satu fungsi yang bisa digunakan dalam spesifikasi. Namun ,tidak semua fungsi bisa digunakan sebagai *measure*. Ada banyak restriksi yang harus dipenuhi untuk menjadikan fungsi sebagai sebuah *measure*, salah satunya adalah fungsi tersebut harus hanya memiliki satu parameter (Vazou et al., 2014). Salah satu contoh fungsi *measure* adalah fungsi warna berikut ini:

```
{-@ measure color @-}
color :: RBSet a -> Color
color (T c _ _ _) = c
```

```
color E = B
color EE = BB
```

Fungsi ini bisa membantu fungsi yang memiliki spesifikasi yang berkaitan dengan warna simpul akar dari sebuah pohon. Contoh penggunaan fungsi *measure* tersebut adalah seperti yang digunakan pada *precondition* fungsi `redden` berikut ini:

```
{-@ redden :: {x:RBSet a | color x == B} -> RBSet a @-}
redden :: RBSet a -> RBSet a
redden (T _ x a b) = T R x a b
```

Fungsi `color` membantu penulisan *precondition* fungsi `redden` bahwa fungsi ini hanya menerima masukan pohon dengan simpul akar berwarna hitam. Hal ini akan sangat sulit dilakukan tanpa menggunakan *measure* tersebut.

Ada satu lagi fitur untuk menyingkat penulisan spesifikasi bernama *type*. Berbeda dengan *inline* yang dapat membantu menyingkat penulisan kondisi, *type* dalam mebantu menyingkat penulisan tipe. Contoh dari penulisan *type* adalah seperti berikut:

```
{-@ type TL a X = Tree {v:a | v < X}  @-}
{-@ type TR a X = Tree {v:a | X < v}  @-}
```

Penulisan *type* sebagai mana dituliskan pada contoh ini dapat membantu mempersingkat penulisan spesifikasi struktur data Tree yang sudah dituliskan sebelumnya menjadi seperti berikut:

```
{-@ data Tree a =
                    E
                  | T { key   :: a
                      , lt    :: TL a key
                      , rt    :: TR a key
                      }
@-}
```

Seperti semua fitur penyingkat lainnya, fitur ini akan sangat membantu jika ada sebuah kondisi yang panjang yang dituliskan berulang-ulang sehingga proses penyingkatan akan sangat mengurangi usaha yang dibutuhkan untuk menuliskan spesifikasi.

II.3.2 Verifikasi Program

Verifikasi program dilakukan dengan menjalankan program Liquid Haskell yang sudah terinstalasi dan memberikan nama dan lokasi *file* yang akan diverifikasi sebagai argumen. Program akan memberikan hasil berupa `SAFE` jika program tersebut lolos verifikasi ataupun `UNSAFE` beserta dengan deskripsi letak kesalahan program jika program tersebut gagal dalam verifikasi.

BAB III

RENCANA PENYELESAIAN MASALAH

III.1 Analisis Masalah

III.1.1 Sumber Kode yang Akan Diverifikasi

Might (Might, 2010) menuliskan bahwa dia membuat implementasi program yang dia tulis dengan menggunakan bahasa Racket. Kemudian, dia menulis ulang program tersebut dalam bahasa Haskell sembari menambahkan beberapa kode dalam *file* lain untuk melakukan verifikasi dengan bantuan *library* QuickCheck. Kemudian, Weirich (Weirich, 2014) mengimplementasikan kembali algoritma tersebut dengan sistem tipe yang jauh lebih ketat dengan menggunakan GADT. Dalam tempat yang sama, Weirich juga menggabungkan kode Haskell asli Might dan kode verifikasi QuickCheck dalam satu *file* dan menambahkan beberapa modifikasi yang membuat kode menjadi sedikit lebih mudah dipahami. Menurut penulis, kode versi ini adalah kode yang paling baik untuk menjadi kode target verifikasi karena kode ini merupakan kode yang paling memudahkan seorang untuk melakukan verifikasi langsung di satu tempat serta memiliki beberapa fungsi tambahan yang bisa membantu dalam memahami maksud penulisan program. Kode ini juga memiliki beberapa ekuivalensi dengan kode GADT yang ditulis Weirich sehingga seseorang bisa mendapat petunjuk tambahan atas apa yang program tersebut lakukan dengan melihat juga kode GADT yang sudah diimplementasikan. Sumber kode tersebut dapat dilihat pada https://github.com/sweirich/dth/blob/master/examples/red-black/MightRedBlack.hs dan sudah terlampirkan dalam Lampiran A.

III.1.2 Sumber Penentuan Spesifikasi

Ranjit Jhala menyatakan bahwa tidak ada program yang bisa dikatakan benar (Jhala, 2018). Hanya ada program yang berhasil mematuhi spesifikasi yang sudah

diberikan kepada program tersebut. Karena itu, sebelum verifikasi dilakukan terhadap sebuah program, harus ada spesifikasi yang terlebih dahulu ditentukan untuk menjadi patokan kebenaran dari implementasi sebuah program. Jika seseorang tidak memiliki kontak dengan penulis program, maka akan terjadi kesulitan untuk menentukan apakah masukan dan keluaran yang sebenarnya diinginkan oleh penulis program saat menulis suatu fungsi. Namun, ada beberapa petunjuk yang bisa dilihat untuk menentukan spesifikasi seperti apa yang cocok disematkan dalam sebuah fungsi dalam program.

III.1.2.1 Tiga Properti RBT

Spesifikasi yang paling penting untuk diverifikasi dalam setiap implementasi struktur data RBT adalah tiga properti yang harus dipatuhi oleh setiap RBT. Setiap fungsi yang bisa digunakan oleh pengguna harus mendapat masukan RBT yang valid dan akan memberikan keluaran RBT yang valid. Jadi, untuk setiap fungsi yang bisa digunakan oleh pengguna minimal harus memiliki 3 spesifikasi sebagai berikut:

1. Fungsi akan menerima dan menghasilkan pohon yang untuk setiap simpul, setiap nilai pada cabang sebelah kiri memiliki nilai lebih kecil dari nilai pada simpul tersebut dan setiap nilai pada cabang sebelah kanan memiliki nilai lebih besar daripada nilai pada simpul tersebut (Properti BST).
2. Fungsi akan menerima dan menghasilkan RBT yang tidak memiliki simpul merah yang memiliki simpul anak berwarna merah (Properti Warna).
3. Fungsi akan menerima dan menghasilkan RBT yang memiliki jumlah simpul hitam yang sama untuk setiap jalur dari simpul akar ke simpul anak (Properti Tinggi).

Ketiga spesifikasi tersebut harus ada dalam setiap fungsi tersebut bersamaan dengan spesifikasi lain yang mungkin harus dipenuhi oleh fungsi tersebut.

III.1.2.2 Spesifikasi Literatur Sebelumnya untuk Fungsi Perantara

Tidak semua fungsi dalam implementasi struktur data RBT selalu mematuhi ketiga properti yang sudah disebutkan sebelumnya. Ada beberapa fungsi perantara yang

digunakan oleh program dan tidak bisa diakses oleh pengguna. Fungsi-fungsi ini tidak didesain untuk mematuhi ketiga properti tersebut untuk memudahkan pemrograman. Namun, pada akhirnya saat fungsi-fungsi yang memanfaatkan fungsi-fungsi perantara tersebut harus tetap mampu menghasilkan RBT yang valid.

Dengan demikian, spesifikasi seperti apa yang bisa diberikan untuk fungsi-fungsi perantara tersebut? Salah satu contoh dari spesifikasi lain tersebut diberikan oleh (Vazou et al., 2014) yang memberikan sebuah properti *almostRB*. Properti ini menyatakan sebuah RBT yang mematuhi seluruh properti lain kecuali properti warna pada simpul paling atas atau simpul akar. Hal ini berarti bahwa simpul akar berwarna merah mungkin memiliki simpul anak berwarna merah juga. Beberapa fungsi perantara memiliki keluaran pohon yang bersifat seperti ini. Karena itu, properti ini bisa digunakan untuk menjadi spesifikasi keluaran yang harus diberikan oleh fungsi-fungsi tersebut.

Implementasi RBT secara GADT milik Weirich (Weirich, 2014) memiliki empat tipe yang masing-masing merupakan tipe yang harus dipatuhi oleh fungsi-fungsi yang menggunakannya. Keempat tipe tersebut adalah:

1. RBSet; Tipe ini menandakan RBT yang valid
2. CT (*Constructed Tree*); Tipe ini menandakan RBT yang melanggar properti empat yang mengharuskan simpul akar memiliki warna hitam.
3. IR (*Intermediate*); Tipe ini menandakan RBT yang memiliki simpul akar yang mungkin melanggar properti warna. Ekuivalen dengan *almostRB*.
4. DT (*Deletion Tree*); Tipe ini menandakan RBT yang mungkin memiliki warna BB atau NB pada simpul daun atau akar. Tipe ini ekuivalen dengan tipe IR yang ditambahkan keterangan bahwa simpul daun dan akar mungkin memiliki simpul berwarna BB dan NB,

Implementasi GADT Weirich menggunakan keempat tipe tersebut sebagai masukan dan keluaran untuk setiap fungsi yang berada dalam program tersebut baik itu fungsi yang akan digunakan oleh pengguna ataupun fungsi perantara. Karena itu, hal ini bisa memberi petunjuk untuk penetapan spesifikasi untuk setiap fungsi perantara yang berada dalam implementasi yang akan diverifikasi.

III.1.3 Modifikasi Program

Untuk membuktikan bahwa Liquid Haskell mampu untuk menangani program yang ditulis oleh orang lain secara independen, sangat penting untuk tidak mengubah program sama sekali saat verifikasi. Hal ini juga dilakukan untuk menyimulasi verifikasi sebuah kode yang tidak bisa diubah karena alasan-alasan tertentu seperti tidak adanya hak atau akses untuk mengubah program tersebut. Namun, jika ditemukan bahwa verifikasi tanpa modifikasi program tidak mungkin dilakukan, maka penulis akan mencoba untuk melakukan modifikasi seminimal mungkin dan menganalisis apa saja perubahan minimal yang harus dilakukan agar program menjadi bisa diverifikasi oleh Liquid Haskell.

III.2 Rencana Penyelesaian Masalah

III.2.1 Tahap Penulisan Spesifikasi

Ada beberapa langkah yang bisa dilakukan untuk menuliskan spesifikasi yang sesuai dengan program ini.

III.2.1.1 Gunakan *precondition* untuk mencegah eksekusi fungsi `error` (*exception*)

Pada beberapa baris program ada yang menggunakan fungsi `error` dalam satu cabang. Fungsi ini digunakan untuk menandakan bahwa suatu fungsi menerima masukan yang tidak diinginkan sehingga menyebabkan program berhenti bekerja. Fungsi ini merupakan fungsi yang ekuivalen dengan fitur *exception* pada bahasa pemrograman lain. Liquid Haskell akan menyatakan bahwa seluruh fungsi yang mungkin menjalankan fungsi `error` tersebut tidak lolos verifikasi. Hal ini akan terjadi kecuali jika *precondition* disesuaikan agar fungsi tersebut tidak akan pernah menerima masukan yang akan membuat program menjalankan fungsi `error` tersebut. Salah satu contoh paling sederhana adalah fungsi pembagian berikut ini:

```
div :: (Fractional a) => a -> a -> a
div x 0 = error "Pembagian dengan angka 0"
div x y = x / y
```

Fungsi pembagian ini akan dianggap gagal verifikasi karena fungsi ini mungkin akan menerima masukan 0 sebagai penyebut. Namun jika fungsi tersebut diberikan *precondition* sebagaimana berikut:

```
{-@ div :: (Fractional a) => a -> {y:a | y /= 0} -> a @-}
div :: (Fractional a) => a -> a -> a
div x 0 = error "Pembagian dengan angka 0"
div x y = x / y
```

Maka program tidak akan pernah menerima masukan 0 sebagai penyebut sehingga program akan lolos tahap verifikasi.

III.2.1.2 Tuliskan fungsi untuk spesifikasi tiga properti RBT

Seluruh properti RBT harus dituliskan kembali sebagai sebuah fungsi yang akan menjadi *measure* atau *inline* untuk membantu verifikasi. Kemudian, seluruh fungsi itu bisa direferensi dan digabung dalam sebuah fungsi lain untuk menyatakan bahwa sebuah pohon adalah RBT yang valid. Fungsi itu bisa berisi hanya bahwa sebuah pohon memenuhi properti BST, properti warna dan properti tinggi.

Namun ada juga fungsi-fungsi gabungan lain yang mungkin dibutuhkan. Misalnya, 4 tipe yang digunakan oleh Weirich bisa dikonversi menjadi fungsi yang merupakan gabungan dari fungsi tiga properti RBT yang sudah dituliskan. Fungsi gabungan yang disebutkan sebelumnya merupakan fungsi yang ekuivalen dengan tipe CT. Tipe RBSet bisa dikonversi menjadi fungsi yang berisi fungsi CT tersebut dan juga syarat bahwa akar pohon berwarna hitam. Tipe DT dan IR bisa menjadi fungsi yang hanya berisi properti BST, properti tinggi, dan syarat bahwa properti warna dipenuhi namun hanya pada semua simpul selain dari simpul akar dan anaknya.

III.2.1.3 Tulis semua fungsi pembantu yang diperlukan

Beberapa properti yang akan diangkat menjadi verifikasi buat merupakan sebuah properti sederhana yang bisa diimplementasikan hanya dalam satu fungsi. Karena itu, dibutuhkan beberapa fungsi lain untuk menjadi pembantu untuk menuliskan properti-properti tersebut menjadi fungsi yang sesuai.

III.2.1.4 Tulis *precondition* dan *postcondition* yang sesuai untuk semua fungsi

Pada akhirnya saat semua properti yang dibutuhkan sudah diimplementasikan sebagai fungsi, maka fungsi-fungsi itu harus digunakan sebagai *precondition* dan *postcondition* fungsi yang membutuhkannya. Salah satu hal yang bisa dilakukan dalam tahap ini adalah perlakuan konversi kode GADT Weirich menjadi kode spesifikasi Liquid Haskell. Salah satu contohnya adalah pada kode Weirich fungsi `ins` memiliki *signature* sebagai berikut:

```
ins :: Ord a => a -> CT n c a -> IR n a
```

Kode ini bisa diubah menjadi kode spesifikasi LH sebagai berikut

```
{-@ ins :: (Ord a) => a
                   -> x:CT a
                   -> {v:IM a | blackHeightL v == blackHeightL x}
@-}
```

Kode ini menyatakan bahwa fungsi `ins` menerima masukan sebuah nilai dan sebuah RBT yang memenuhi tipe CT. Kemudian fungsi ini akan memberikan keluaran sebuah RBT yang memenuhi tipe IM dan memiliki tinggi simpul hitam yang sama dengan tinggi simpul hitam pada masukan.

III.2.2 Perangkat untuk Verifikasi

Penulisan kode dan spesifikasi bisa dilakukan di perangkat lunak penulis teks apa pun. Untuk melakukan verifikasi, seseorang harus terlebih dahulu melakukan instalasi ghc 8.6.5 pada komputer yang dimiliki. Kemudian instalasi cabal juga diperlukan untuk membantu instalasi beberapa *library* Haskell. Setelah itu, seseorang bisa menggunakan perangkat lunak cabal untuk melakukan instalasi LiquidHaskell-0.8.10.2. Untuk verifikasi juga dibutuhkan instalasi sebuah *SMT Solver*. Untuk penulis, *SMT Solver* yang bisa bekerja dengan baik pada komputer penulis adalah Z3 4.8.7. Jika semua perangkat lunak tersebut sudah selesai diinstalasi, maka verifikasi bisa dilakukan dengan mudah dengan menjalankan perintah `liquid <nama dan lokasi file>` dan *file* tersebut akan diverifikasi oleh Liquid Haskell.

Jika seseorang tidak mau atau tidak mampu untuk melakukan hal tersebut, verifikasi sebuah kode juga bisa dilakukan secara daring pada laman http://goto.ucsd.edu:8090/index.html atau https://liquid-demo.programming.systems/index.html . Seorang pengguna hanya perlu menuliskan kode yang ingin diverifikasi pada tempat yang disediakan kemudian menekan tombol “Re Check”. Pengguna juga bisa menyimpan kode tersebut dengan menekan tombol “Permalink” untuk bisa melihat kembali kode tersebut di kemudian hari atau untuk membagikan kode tersebut pada orang lain. Pada laman tersebut juga ada beberapa kode contoh yang bisa digunakan untuk melihat contoh-contoh kode yang bisa diverifikasi menggunakan Liquid Haskell.

DAFTAR PUSTAKA

Baier, C., & Katoen, J.-P. (2008). Principles Of Model Checking. In *MIT Press*.

Bayer, R. (1972). Symmetric binary B-Trees: Data structure and maintenance algorithms. *Acta Informatica*. https://doi.org/10.1007/BF00289509

Germane, K., & Might, M. (2014). Deletion: The curse of the red-black tree. *Journal of Functional Programming*, *24*(4), 423–433. https://doi.org/10.1017/S0956796814000227

Holloway, C. M. (1997). Why engineers should consider formal methods. *AIAA/IEEE Digital Avionics Systems Conference - Proceedings*, *1*(October). https://doi.org/10.1109/dasc.1997.635021

Huth, M., & Ryan, M. (2004). *Logic in CS: Modelling and Reasoning about Systems* (Vol. 66, Issue 1).

Jhala, R. (2018). *The Hillelogram Verifier Rodeo I (LeftPad)*. https://ucsd-progsys.github.io/liquidhaskell-blog/2018/05/17/hillel-verifier-rodeo-I-leftpad.lhs/

Kahrs, S. (2001). Red-black trees with types. *Journal of Functional Programming*, *11*(4), 425–432. https://doi.org/10.1017/S0956796801004026

Might, M. (2010). *The missing method: Deleting from Okasaki's red-black trees*. http://matt.might.net/articles/red-black-delete/

Monin, J.-F. (2003). Understanding Formal Methods. In *Understanding Formal Methods*. https://doi.org/10.1007/978-1-4471-0043-0

Okasaki, C. (1999). Functional pearl: Red-black trees in a functional setting. *Journal of Functional Programming*, *9*(4), 471–477. https://doi.org/10.1017/S0956796899003494

Peña, R. (2017). An introduction to liquid Haskell. *Electronic Proceedings in Theoretical Computer Science, EPTCS*, *237*, 68–80. https://doi.org/10.4204/EPTCS.237.5

Sedgewick, R. (2008). Left-leaning Red-Black Trees. *Public Talk*, 8. http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf

Vazou, N., Lampropoulos, L., & Polakow, J. (2017). A tale of two provers: Verifying monoidal string matching in liquid Haskell and Coq. *ACM SIGPLAN Notices*, *52*(10), 63–74. https://doi.org/10.1145/3122955.3122963

Vazou, N., Seidel, E. L., & Jhala, R. (2014). LiquidHaskell: Experience with refinement types in the real world. *Haskell 2014 - Proceedings of the 2014 ACM SIGPLAN Haskell Symposium*, *2*, 39–51. https://doi.org/10.1145/2633357.2633366

Weirich, S. (2014). *Implementation of deletion for red black trees by Matt Might; Editing to preserve the red/black tree invariants by Stephanie Weirich, Dan Licata and John Hughes*. https://github.com/sweirich/dth/blob/master/examples/red-black/MightRedBlackGADT.hs

Lampiran A. Sumber Kode Implementasi Algoritma Penghapusan RBT Might

```
1. -- Implementation of deletion for red black trees by Matt
   Might
2.
3. -- Original available from:
4. --   http://matt.might.net/articles/red-black-
   delete/code/RedBlackSet.hs
5. -- Slides:
6. --   http://matt.might.net/papers/might2014redblack-talk.pdf
7. -- Draft paper:
8. --   http://matt.might.net/tmp/red-black-pearl.pdf
9.
10. module MightRedBlack where
11.
12. import Prelude hiding (max)
13. import Control.Monad
14. import Test.QuickCheck hiding (elements)
15. import Data.List(nub,sort)
16.
17. data Color =
18.    R  -- red
19.  | B  -- black
20.  | BB -- double black
21.  | NB -- negative black
22.  deriving (Show, Eq)
23.
24. data RBSet a =
25.    E  -- black leaf
26.  | EE -- double black leaf
27.  | T Color (RBSet a) a (RBSet a)
28.  deriving (Show, Eq)
29.
30.  -- Private auxiliary functions --
31.
32. redden :: RBSet a -> RBSet a
33. redden (T _ a x b) = T R a x b
34.
35. -- blacken for insert
36. -- never a leaf, could be red or black
37. blacken' :: RBSet a -> RBSet a
38. blacken' (T R a x b) = T B a x b
39. blacken' (T B a x b) = T B a x b
40.
41. -- blacken for delete
42. -- root is never red, could be double black
43. blacken :: RBSet a -> RBSet a
44. blacken (T B a x b) = T B a x b
45. blacken (T BB a x b) = T B a x b
46. blacken E = E
47. blacken EE = E
48.
49. isBB :: RBSet a -> Bool
```

```
50.  isBB EE = True
51.  isBB (T BB _ _ _) = True
52.  isBB _ = False
53.
54.  blacker :: Color -> Color
55.  blacker NB = R
56.  blacker R = B
57.  blacker B = BB
58.  blacker BB = error "too black"
59.
60.  redder :: Color -> Color
61.  redder NB = error "not black enough"
62.  redder R = NB
63.  redder B = R
64.  redder BB = B
65.
66.  blacker' :: RBSet a -> RBSet a
67.  blacker' E = EE
68.  blacker' (T c l x r) = T (blacker c) l x r
69.
70.  redder' :: RBSet a -> RBSet a
71.  redder' EE = E
72.  redder' (T c l x r) = T (redder c) l x r
73.
74.   -- `balance` rotates away coloring conflicts:
75.  balance :: Color -> RBSet a -> a -> RBSet a -> RBSet a
76.
77.   -- Okasaki's original cases:
78.  balance B (T R (T R a x b) y c) z d = T R (T B a x b) y (T
    B c z d)
79.  balance B (T R a x (T R b y c)) z d = T R (T B a x b) y (T
    B c z d)
80.  balance B a x (T R (T R b y c) z d) = T R (T B a x b) y (T
    B c z d)
81.  balance B a x (T R b y (T R c z d)) = T R (T B a x b) y (T
    B c z d)
82.
83.   -- Six cases for deletion:
84.  balance BB (T R (T R a x b) y c) z d = T B (T B a x b) y
    (T B c z d)
85.  balance BB (T R a x (T R b y c)) z d = T B (T B a x b) y
    (T B c z d)
86.  balance BB a x (T R (T R b y c) z d) = T B (T B a x b) y
    (T B c z d)
87.  balance BB a x (T R b y (T R c z d)) = T B (T B a x b) y
    (T B c z d)
88.
89.  balance BB a x (T NB (T B b y c) z d@(T B _ _ _))
90.      = T B (T B a x b) y (balance B c z (redden d))
91.  balance BB (T NB a@(T B _ _ _) x (T B b y c)) z d
92.      = T B (balance B (redden a) x b) y (T B c z d)
93.
94.  balance color a x b = T color a x b
95.
96.   -- `bubble` "bubbles" double-blackness upward toward the
    root:
```

```
97.  bubble :: Color -> RBSet a -> a -> RBSet a -> RBSet a
98.  bubble color l x r
99.    | isBB(l) || isBB(r) = balance (blacker color) (redder'
   l) x (redder' r)
100.   | otherwise          = balance color l x r
101.
102.
103.
104.
105.  -- Public operations --
106.
107. empty :: RBSet a
108. empty = E
109.
110.
111. member :: (Ord a) => a -> RBSet a -> Bool
112. member x E = False
113. member x (T _ l y r) | x < y     = member x l
114.                     | x > y     = member x r
115.                     | otherwise = True
116.
117.
118. insert :: (Ord a) => a -> RBSet a -> RBSet a
119. insert x s = blacken' (ins s)
120.  where ins E = T R E x E
121.        ins s@(T color a y b) | x < y     = balance color
   (ins a) y b
122.                             | x > y     = balance color a
   y (ins b)
123.                             | otherwise = s
124.
125.
126. max :: RBSet a -> a
127. max E = error "no largest element"
128. max (T _ _ x E) = x
129. max (T _ _ x r) = max r
130.
131. -- Remove this node: it might leave behind a double black
   node
132. remove :: RBSet a -> RBSet a
133. -- remove E = E   -- impossible!
134. -- ; Leaves are easiest to kill:
135. remove (T R E _ E) = E
136. remove (T B E _ E) = EE
137. -- ; Killing a node with one child;
138. -- ; parent or child is red:
139. -- remove (T R E _ child) = child
140. -- remove (T R child _ E) = child
141. remove (T B E _ (T R a x b)) = T B a x b
142. remove (T B (T R a x b) _ E) = T B a x b
143. -- ; Killing a black node with one black child:
144. -- remove (T B E _ child@(T B _ _ _)) = blacker' child
145. -- remove (T B child@(T B _ _ _) _ E) = blacker' child
146. -- ; Killing a node with two sub-trees:
147. remove (T color l y r) = bubble color l' mx r
148.  where mx = max l
```

```
149.         l' = removeMax l
150.
151. removeMax :: RBSet a -> RBSet a
152. removeMax E = error "no maximum to remove"
153. removeMax s@(T _ _ _ E) = remove s
154. removeMax s@(T color l x r) = bubble color l x (removeMax
   r)
155.
156. delete :: (Ord a) => a -> RBSet a -> RBSet a
157. delete x s = blacken (del x s)
158.
159. del x E = E
160. del x s@(T color a' y b') | x < y   = bubble color (del x
   a') y b'
161.                          | x > y     = bubble color a' y
   (del x b')
162.                          | otherwise = remove s
163.
164. prop_del :: Int -> RBSet Int -> Bool
165. prop_del x s = color (del x s) `elem` [B, BB]
166.
167.
168. --- Testing code
169.
170. elements :: Ord a => RBSet a -> [a]
171. elements t = aux t [] where
172.       aux E acc = acc
173.       aux (T _ a x b) acc = aux a (x : aux b acc)
174.
175. instance (Ord a, Arbitrary a) => Arbitrary (RBSet a)
   where
176.   arbitrary = liftM (foldr insert empty) arbitrary
177.
178. prop_BST :: RBSet Int -> Bool
179. prop_BST t = isSortedNoDups (elements t)
180.
181. color :: RBSet a -> Color
182. color (T c _ _ _) = c
183. color E = B
184. color EE = BB
185.
186. prop_Rb2 :: RBSet Int -> Bool
187. prop_Rb2 t = color t == B
188.
189. prop_Rb3 :: RBSet Int -> Bool
190. prop_Rb3 t = fst (aux t) where
191.   aux E = (True, 0)
192.   aux (T c a x b) = (h1 == h2 && b1 && b2, if c == B then
   h1 + 1 else h1) where
193.       (b1 , h1) = aux a
194.       (b2 , h2) = aux b
195.
196. prop_Rb4 :: RBSet Int  -> Bool
197. prop_Rb4 E = True
198. prop_Rb4 (T R a x b) = color a == B && color b == B &&
   prop_Rb4 a && prop_Rb4 b
```

```
199. prop_Rb4 (T B a x b) = prop_Rb4 a && prop_Rb4 b
200.
201.
202. isSortedNoDups :: Ord a => [a] -> Bool
203. isSortedNoDups x = nub (sort x) == x
204.
205.
206. prop_delete_spec1 :: RBSet Int -> Bool
207. prop_delete_spec1 t = all (\x -> not (member x (delete x
   t))) (elements t)
208.
209. prop_delete_spec2 :: RBSet Int -> Bool
210. prop_delete_spec2 t = all (\(x,y) -> x == y || (member y
   (delete x t))) allpairs where
211.    allpairs = [ (x,y) | x <- elements t, y <- elements t ]
212.
213. prop_delete_spec3 :: RBSet Int -> Int -> Property
214. prop_delete_spec3 t x = not (x `elem` elements t) ==>
   (delete x t == t)
215.
216. prop_delete_bst :: RBSet Int -> Bool
217. prop_delete_bst t = all (\x -> prop_BST (delete x t))
   (elements t)
218.
219. prop_delete2 :: RBSet Int -> Bool
220. prop_delete2 t = all (\x -> prop_Rb2 (delete x t))
   (elements t)
221.
222. prop_delete3 :: RBSet Int -> Bool
223. prop_delete3 t = all (\x -> prop_Rb3 (delete x t))
   (elements t)
224.
225. prop_delete4 :: RBSet Int -> Bool
226. prop_delete4 t = all (\x -> prop_Rb4 (delete x t))
   (elements t)
227.
228. check_insert = do
229.     putStrLn "BST property"
230.     quickCheck prop_BST
231.     putStrLn "Root is black"
232.     quickCheck prop_Rb2
233.     putStrLn "Black height the same"
234.     quickCheck prop_Rb3
235.     putStrLn "Red nodes have black children"
236.     quickCheck prop_Rb4
237.
238. check_delete = do
239.    quickCheckWith (stdArgs {maxSuccess=100})
   prop_delete_spec1
240.    quickCheckWith (stdArgs {maxSuccess=100})
   prop_delete_spec2
241.    quickCheckWith (stdArgs {maxSuccess=100})
   prop_delete_spec3
242.    quickCheckWith (stdArgs {maxSuccess=100}) prop_delete2
243.    quickCheckWith (stdArgs {maxSuccess=100}) prop_delete3
244.    quickCheckWith (stdArgs {maxSuccess=100}) prop_delete4
```

```
245.    quickCheckWith (stdArgs {maxSuccess=100})
   prop_delete_bst
246.
247.
248. main :: IO ()
249. main =
250.  do
251.  return $! ()
```