

BAB II

STUDI LITERATUR

II.1 Liquid Haskell

Liquid Haskell adalah implementasi teknologi *Liquid Types* (Rondon et al., 2008) yang digunakan untuk membantu memverifikasi sebuah program berbahasa Haskell (Peña, 2017). *Liquid Types* atau *Logically Qualified Data Types* adalah sebuah sistem yang mengombinasikan inferensi tipe *Hindley-Milner* dengan *Predicate Abstraction* untuk menghasilkan secara inferensi tipe secara otomatis yang bisa dicek secara mudah oleh *SMT Solver* otomatis seperti Z3 atau CVC4. Liquid Haskell pada dasarnya melakukan konversi sebuah program Haskell menjadi *Liquid Types* yang sesuai dengan karakteristik program untuk kemudian diberikan pada *SMT Solver* untuk dicek kebenarannya. Jika terjadi inkonsistensi pada *Liquid Types* yang diterima maka bisa diputuskan bahwa program Haskell yang diberikan memiliki *bug* yang mampu merusak konsistensi program.

II.1.1 Penggunaan Liquid Haskell

II.1.1.1 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. Aturan sintaksis lengkap dari Liquid Haskell dapat dilihat pada Gambar II.1.

e	$::=$	x c $\lambda x.e$ $e e$ $\text{if } e \text{ then } e \text{ else } e$ $\text{let } x = e \text{ in } e$ $\text{let rec } f = \lambda x.e \text{ in } e$ $[\Lambda\alpha]e$ $[\tau]e$	<i>Expressions:</i> variable constant abstraction application if-then-else let-binding letrec-binding type-abstraction type-instantiation
Q	$::=$	true q $Q \wedge Q$	<i>Liquid Refinements</i> true logical qualifier in Q^* conjunction of qualifiers
B	$::=$	int bool	<i>Base Types:</i> base type of integers base type of booleans
$T(B)$	$::=$	$\{\nu : B \mid B\}$ $x : T(B) \rightarrow T(B)$ α	<i>Type Skeletons:</i> base function type variable
$S(B)$	$::=$	$T(B)$ $\forall \alpha. S(B)$	<i>Type Schema Skeletons:</i> monotype polytype
τ, σ	$::=$	$T(\text{true}), S(\text{true})$	<i>Types, Schemas</i>
T, S	$::=$	$T(E), S(E)$	<i>Dep. Types, Schemas</i>
\hat{T}, \hat{S}	$::=$	$T(Q), S(Q)$	<i>Liquid Types, Schemas</i>

Gambar II.1. Sintaksis Liquid Haskell (Rondon et al., 2008)

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 sehingga 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.1.1.1.1 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 *redde*n berikut ini:

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

Fungsi *color* membantu penulisan *precondition* fungsi *redde*n sehingga 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 membantu 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.1.1.1.2 Spesifikasi untuk Fungsi yang digunakan oleh Fungsi Lain

Fungsi-fungsi pada Haskell didesain untuk bisa beroperasi saling independen. Setiap fungsi bisa digunakan oleh pengguna tanpa perlu memedulikan apa pun tentang fungsi lain yang berada di *file* lain atau bahkan dalam *file* yang sama. Setiap fungsi juga dapat menggunakan fungsi lain sesuai dengan kebutuhan yang diperlukan oleh fungsi tersebut. Namun, fungsi yang digunakan tersebut bisa saja mengalami perubahan secara independen yang membuat fungsi itu tidak cocok lagi digunakan untuk fungsi yang menggunakannya. Dengan kata lain, setiap fungsi memiliki sifat seperti kotak hitam terhadap fungsi lainnya yang tidak mengetahui apa yang sebenarnya dilakukan oleh fungsi yang digunakan tersebut. Fungsi pengguna hanya bisa mengharapkan bahwa fungsi yang digunakan mematuhi logika tertentu yang penting untuk digunakan oleh fungsi pengguna. Untuk itu, dituliskan spesifikasi khusus pada fungsi yang digunakan oleh fungsi lain untuk menjamin bahwa fungsi tersebut bisa digunakan oleh fungsi pengguna.

Kita dapat menggunakan fungsi *inverse* yang berada pada II.1.1.1 sebagai contoh. Dibuat sebuah fungsi sederhana $\text{operasiA } x \ y = x * 2 / (\text{inverse } y)$. Fungsi *operasiA* menggunakan operasi bagi (/) yang membutuhkan penyebut yang tidak bernilai 0. Jika fungsi *inverse* memiliki

kemungkinan untuk menghasilkan nilai sama dengan 0, maka fungsi operasiA memiliki kemungkinan mendapatkan *error division by zero* yang berarti fungsi tersebut tidak konsisten. Untungnya, fungsi inverse sudah memiliki spesifikasi `{-@ inverse :: {x:Int | x > 0} -> {v:Int | v < 0} @-}` yang berarti keluaran dari fungsi ini dijamin memiliki nilai yang lebih kecil dari 0. Tidak ada nilai yang lebih kecil dari 0 memiliki nilai yang sama dengan 0, sehingga dengan ini fungsi operasiA dianggap konsisten. Namun tentu saja, dalam saat yang sama fungsi inverse memiliki *precondition* yang mensyaratkan bahwa fungsi ini mendapatkan masukan yang selalu lebih besar dari 0. Untuk itu, fungsi operasiA bisa diubah sehingga fungsi tersebut pasti akan memberikan nilai yang lebih besar dari 0 atau fungsi tersebut dapat juga memiliki spesifikasi yang mensyaratkan bahwa nilai yang diterima fungsi tersebut lebih besar dari 0. Contoh spesifikasi tersebut dapat ditulis sebagaimana berikut ini `{-@ operasiA :: Int -> {y:Int | y > 0} -> Int @-}`.

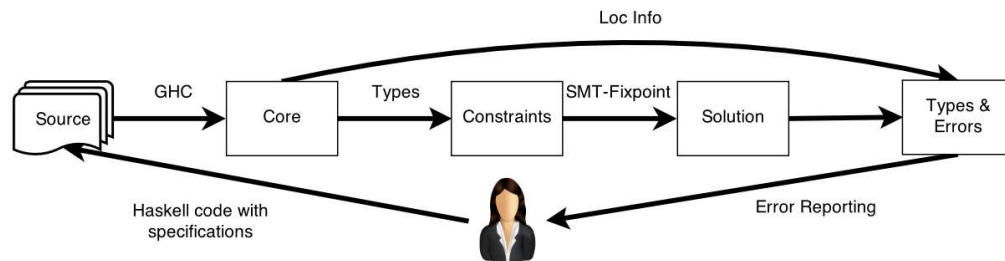
Jika fungsi operasiA kemudian dibutuhkan oleh fungsi lain yang mensyaratkan properti tertentu, maka spesifikasi ini bisa diubah sehingga spesifikasi ini akan mensyaratkan keluaran yang dibutuhkan tersebut. Tentu saja, fungsi ini harus tetap konsisten dan lulus dalam pengecekan konsistensi terhadap kondisi keluaran yang baru tersebut. Misalkan ada fungsi lain yang membutuhkan operasiA dan mensyaratkan bahwa hasil operasiA selalu genap. Maka spesifikasi bisa diubah sehingga menjadi `{-@ operasiA :: Int -> {y:Int | y > 0} -> {v:Int | v % 2 == 0} @-}`. Kemudian ada fungsi lain yang membutuhkan operasiA memiliki nilai yang lebih kecil dari 0. Spesifikasi operasiA dapat diubah menjadi `{-@ operasiA :: Int -> {y:Int | y > 0} -> {v:Int | (v % 2 == 0) && (v < 0)} @-}`. Namun spesifikasi ini tidak akan diterima oleh Liquid Haskell karena operasiA belum tentu menghasilkan nilai yang hanya positif. Oleh karena itu, berarti terjadi inkonsistensi dalam program yang harus dicek kembali

II.1.1.2 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.

II.1.2 Alur Kerja Liquid Haskell

Cara kerja Liquid Haskell secara sederhana telah dirangkum pada Gambar II.2.



Gambar II.2. Liquid Haskell *Workflow* (Vazou et al., 2014)

II.1.2.1 Program yang diverifikasi (*Source*)

Liquid Haskell dapat menerima *file* yang berisi program Haskell biasa. Program tersebut juga dapat ditambahkan dengan spesifikasi yang sudah dituliskan dalam format yang dikenali oleh Liquid Haskell. Spesifikasi tersebut juga dapat dituliskan dalam *file* terpisah. Jika spesifikasi tersebut dituliskan bersamaan dengan kode program, maka spesifikasi tersebut dituliskan dalam blok komentar `{-@ @-}` sehingga spesifikasi tersebut tidak akan mengganggu kompilasi program menjadi program Haskell biasa dengan menggunakan kompilator Haskell yang lain.

II.1.2.2 Bahasa Perantara *Core*

Core adalah sebuah bahasa perantara yang digunakan oleh GHC (*Glorious Haskell Compiler*) untuk menyatukan berbagai implementasi bahasa Haskell yang bermacam-macam menjadi sebuah bahasa yang padu dan singkat serta mudah untuk dikompilasi. Penggunaan bahasa *Core* juga memudahkan pembuatan Liquid

Haskell karena bahasa *Core* didesain untuk memudahkan pengecekan tipe yang digunakan oleh kode sehingga dengan menggunakan bahasa *Core* setengah pengecekan tipe yang harus dilakukan oleh Liquid Haskell sudah bisa dikerjakan oleh GHC secara langsung. Oleh karena itu, Liquid Haskell memanfaatkan GHC untuk terlebih dahulu mengonversi kode Haskell yang diterima menjadi kode dalam bahasa *Core* sebelum diproses lebih lanjut oleh Liquid Haskell sendiri. Keuntungan lain dari proses ini adalah dalam proses ini GHC juga mampu mengeliminasi program yang bukan merupakan program Haskell yang valid misalnya karena ada salah ketik dalam program sehingga kode yang akan ditangani oleh Liquid Haskell pasti merupakan kode yang sudah bisa dikompilasi menjadi sebuah program.

II.1.2.3 Kalimat Logika sebagai Batasan (*Constraints*)

Pada tahap ini *constraints* atau batasan diberikan kepada berbagai variabel yang digunakan dalam kode *Core* dalam bentuk *Liquid Types*. Batasan-batasan ini didesain untuk mampu untuk menerangkan karakteristik program dengan menggunakan jumlah kalimat logika yang sesedikit mungkin. Misalnya, untuk kode `y = if x > 0 then x else 1`, salah satu batasan yang bisa diberikan adalah kalimat logika yang menyatakan bahwa `y` pasti memiliki nilai lebih dari 0.

Liquid Types memiliki format `{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. Jadi, misal dari kode `y = if x > 0 then x else 1` yang sudah disebutkan sebelumnya dapat didapatkan sebuah batasan `{y: Int | y > 0}`. Contoh lain adalah kode sederhana bagi `x y = x / y` yang dari kode tersebut dapat diambil batasan `{y: Int | y /= 0}` karena operasi pembagian tidak boleh memiliki penyebut yang bernilai sama dengan nol. Spesifikasi yang dituliskan dalam kode Haskell merupakan serangkaian kalimat logika yang sudah ditulis dalam format *Liquid Types* yang bisa ditambahkan pada

serangkaian *Liquid Types* yang sudah disarikan dari program itu sendiri untuk membantu pembuktian atau menambahkan pengujian konsistensi program terhadap hal tertentu yang diinginkan oleh pengguna.

Sistem dasar dan aturan dari *Liquid Types* dapat dilihat pada Gambar II.3. Aturan ini ditetapkan untuk memastikan bahwa kalimat logika yang dihasilkan mampu diselesaikan oleh *SMT Solver* pada akhirnya (kalimat logika tidak *undecidable*) serta mampu menerangkan berbagai karakteristik dari program seperti percabangan, *polymorphism*, serta pengecekan *subtype* dalam program secara benar (Rondon et al., 2008). Kemudian aturan tersebut akan digunakan untuk memproduksi berbagai *Liquid Types* yang sesuai dengan suatu program dalam algoritma yang ditunjukkan pada Gambar II.4.

Liquid Type Checking

$$\boxed{\Gamma \vdash_{\mathbb{Q}} e : S}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} e : S_1 \quad \Gamma \vdash S_1 <: S_2 \quad \Gamma \vdash S_2}{\Gamma \vdash_{\mathbb{Q}} e : S_2} \text{ [LT-SUB]}$$

$$\frac{\Gamma(x) = \{\nu : B \mid e\}}{\Gamma \vdash_{\mathbb{Q}} x : \{\nu : B \mid \nu = x\}} \text{ [LT-VAR]} \quad \frac{\Gamma(x) \text{ not a base type}}{\Gamma \vdash_{\mathbb{Q}} x : \Gamma(x)} \text{ [LT-VAR]}$$

$$\frac{}{\Gamma \vdash_{\mathbb{Q}} c : ty(c)} \text{ [LT-CONST]}$$

$$\frac{\Gamma; x : \hat{T}_x \vdash_{\mathbb{Q}} e : \hat{T} \quad \Gamma \vdash x : \hat{T}_x \rightarrow \hat{T}}{\Gamma \vdash_{\mathbb{Q}} \lambda x. e : (x : \hat{T}_x \rightarrow \hat{T})} \text{ [LT-FUN]}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} e_1 : (x : T_x \rightarrow T) \quad \Gamma \vdash_{\mathbb{Q}} e_2 : T_x}{\Gamma \vdash_{\mathbb{Q}} e_1 e_2 : [e_2/x]T} \text{ [LT-APP]}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} e_1 : \text{bool} \quad \Gamma; e_1 \vdash_{\mathbb{Q}} e_2 : \hat{T} \quad \Gamma; \neg e_1 \vdash_{\mathbb{Q}} e_3 : \hat{T} \quad \Gamma \vdash \hat{T}}{\Gamma \vdash_{\mathbb{Q}} \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \hat{T}} \text{ [LT-IF]}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} e_1 : S_1 \quad \Gamma; x : S_1 \vdash_{\mathbb{Q}} e_2 : \hat{T} \quad \Gamma \vdash \hat{T}}{\Gamma \vdash_{\mathbb{Q}} \text{let } x = e_1 \text{ in } e_2 : \hat{T}} \text{ [LT-LET]}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} e : S \quad \alpha \text{ not free in } \Gamma}{\Gamma \vdash_{\mathbb{Q}} [\Lambda \alpha]e : \forall \alpha. S} \text{ [LT-GEN]}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} e : \forall \alpha. S \quad \Gamma \vdash \hat{T} \quad \text{Shape}(\hat{T}) = \tau}{\Gamma \vdash_{\mathbb{Q}} [\tau]e : [\hat{T}/\alpha]S} \text{ [LT-INST]}$$

Decidable Subtyping

$$\boxed{\Gamma \vdash S_1 <: S_2}$$

$$\frac{\text{Valid}(\llbracket \Gamma \rrbracket \wedge \llbracket e_1 \rrbracket \Rightarrow \llbracket e_2 \rrbracket)}{\Gamma \vdash \{\nu : B \mid e_1\} <: \{\nu : B \mid e_2\}} \text{ [DEC-}<:-\text{BASE]}$$

$$\frac{\Gamma \vdash T'_x <: T_x \quad \Gamma; x : T'_x \vdash T <: T'}{\Gamma \vdash x : T_x \rightarrow T <: x : T'_x \rightarrow T'} \text{ [DEC-}<:-\text{FUN]}$$

$$\frac{}{\Gamma \vdash \alpha <: \alpha} \text{ [}<:-\text{VAR}] \quad \frac{\Gamma \vdash S_1 <: S_2}{\Gamma \vdash \forall \alpha. S_1 <: \forall \alpha. S_2} \text{ [}<:-\text{POLY}]$$

Well-Formed Types

$$\boxed{\Gamma \vdash S}$$

$$\frac{\Gamma; \nu : B \vdash e : \text{bool}}{\Gamma \vdash \{\nu : B \mid e\}} \text{ [WT-BASE]} \quad \frac{}{\Gamma \vdash \alpha} \text{ [WT-VAR]}$$

$$\frac{\Gamma; x : T_x \vdash T}{\Gamma \vdash x : T_x \rightarrow T} \text{ [WT-FUN]} \quad \frac{\Gamma \vdash S}{\Gamma \vdash \forall \alpha. S} \text{ [WT-POLY]}$$

Gambar II.3. Aturan Pengecekan *Liquid Type* (Rondon et al., 2008)

II.1.2.4 Solusi dan Hasil (*Solution, Types, and Errors*)

Pada akhirnya seluruh batasan yang dihasilkan dari tahap sebelumnya disatukan menjadi *Horn Clause* atau kalimat logika dalam bentuk $u \leftarrow (p \wedge q \wedge \dots \wedge t)$ yang kemudian akan diselesaikan oleh algoritma *fixpoint* yang diterangkan dalam Gambar II.4. Fungsi `solve` pada algoritma tersebut bisa diimplementasikan dengan menggunakan bantuan *SMT Solver* seperti Z3 atau CVC4.

$$\begin{aligned}
\text{Cons}(\Gamma, e) = & \\
& \text{match } e \text{ with} \\
& | x \longrightarrow \\
& \quad \text{if } \text{HM}(\text{Shape}(\Gamma), e) = B \\
& \quad \text{then } (\{\nu : B \mid \nu = x\}, \emptyset) \\
& \quad \text{else } (\Gamma(x), \emptyset) \\
& | c \longrightarrow \\
& \quad (ty(c), \emptyset) \\
& | e_1 e_2 \longrightarrow \\
& \quad \text{let } (x : F_x \rightarrow F, C_1) = \text{Cons}(\Gamma, e_1) \text{ in} \\
& \quad \text{let } (F'_x, C_2) = \text{Cons}(\Gamma, e_2) \text{ in} \\
& \quad ([e_2/x]F, C_1 \cup C_2 \cup \{\Gamma \vdash F'_x <: F_x\}) \\
& | \lambda x. e \longrightarrow \\
& \quad \text{let } (x : F_x \rightarrow F) = \text{Fresh}(\text{HM}(\text{Shape}(\Gamma), \lambda x. e)) \text{ in} \\
& \quad \text{let } (F', C') = \text{Cons}(\Gamma; x : F_x, e) \text{ in} \\
& \quad (x : F_x \rightarrow F, C' \cup \{\Gamma \vdash x : F_x \rightarrow F\} \cup \{\Gamma; x : F_x \vdash F' <: F\}) \\
& | \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \longrightarrow \\
& \quad \text{let } F = \text{Fresh}(\text{HM}(\text{Shape}(\Gamma), e)) \text{ in} \\
& \quad \text{let } (_, C_1) = \text{Cons}(\Gamma, e_1) \text{ in} \\
& \quad \text{let } (F_2, C_2) = \text{Cons}(\Gamma; e_1, e_2) \text{ in} \\
& \quad \text{let } (F_3, C_3) = \text{Cons}(\Gamma; \neg e_1, e_3) \text{ in} \\
& \quad (F, C_1 \cup C_2 \cup C_3 \cup \{\Gamma \vdash F\} \cup \\
& \quad \quad \{\Gamma; e_1 \vdash F_2 <: F\} \cup \{\Gamma; \neg e_1 \vdash F_3 <: F\}) \\
& | \text{let } x = e_1 \text{ in } e_2 \longrightarrow \\
& \quad \text{let } F = \text{Fresh}(\text{HM}(\text{Shape}(\Gamma), e)) \text{ in} \\
& \quad \text{let } (F_1, C_1) = \text{Cons}(\Gamma, e_1) \text{ in} \\
& \quad \text{let } (F_2, C_2) = \text{Cons}(\Gamma; x : F_1, e_2) \text{ in} \\
& \quad (F, C_1 \cup C_2 \cup \{\Gamma \vdash F\} \cup \{\Gamma; x : F_1 \vdash F_2 <: F\})
\end{aligned}$$

$$\begin{aligned}
& | [\Lambda\alpha]e \longrightarrow \\
& \quad \mathbf{let} (F, C) = \mathbf{Cons}(\Gamma, e) \mathbf{in} \\
& \quad (\forall\alpha.F, C) \\
& | [\tau]e \longrightarrow \\
& \quad \mathbf{let} F = \mathbf{Fresh}(\tau) \mathbf{in} \\
& \quad \mathbf{let} (\forall\alpha.F', C) = \mathbf{Cons}(\Gamma, e) \mathbf{in} \\
& \quad ([F/\alpha]F', C \cup \{\Gamma \vdash F\}) \\
\\
& \mathbf{Weaken}(c, A) = \\
& \quad \mathbf{match} c \mathbf{with} \\
& \quad | \Gamma \vdash \{\nu : B \mid \theta \cdot \kappa\} \longrightarrow \\
& \quad \quad A[\kappa \mapsto \{q \mid q \in A(\kappa) \mathbf{and} \mathbf{Shape}(\Gamma); \nu : B \vdash \theta \cdot q : \mathbf{bool}\}] \\
& \quad | \Gamma \vdash \{\nu : B \mid \rho\} <: \{\nu : B \mid \theta \cdot \kappa\} \longrightarrow \\
& \quad \quad A[\kappa \mapsto \{q \mid q \in A(\kappa) \mathbf{and} \llbracket A(\Gamma) \rrbracket \wedge \llbracket A(\rho) \rrbracket \Rightarrow \llbracket \theta \cdot q \rrbracket\}] \\
& \quad | - \longrightarrow \mathbf{Failure} \\
\\
& \mathbf{Solve}(C, A) = \\
& \quad \mathbf{if} \text{ exists } c \in C \text{ such that } A(c) \text{ is not valid} \\
& \quad \mathbf{then} \mathbf{Solve}(C, \mathbf{Weaken}(c, A)) \mathbf{else} A \\
\\
& \mathbf{Infer}(\Gamma, e, \mathbb{Q}) = \\
& \quad \mathbf{let} (F, C) = \mathbf{Cons}(\Gamma, e) \mathbf{in} \\
& \quad \mathbf{let} A = \mathbf{Solve}(\mathbf{Split}(C), \lambda\kappa.\mathbf{Inst}(\Gamma, e, \mathbb{Q})) \mathbf{in} \\
& \quad A(F)
\end{aligned}$$

Gambar II.4. Algoritma *Liquid Type Inference* (Rondon et al., 2008)

Setelah menjalankan algoritma ini, akan dihasilkan inferensi apakah rangkaian batasan yang sudah dihasilkan dari sebuah kode mampu diselesaikan atau tidak. Jika ya, maka Liquid Haskell akan memberikan hasil “SAFE” yang menandakan bahwa kode program telah berhasil diverifikasi dan memiliki logika yang konsisten. Jika tidak, maka Liquid Haskell akan memberikan hasil “UNSAFE” dan menunjukkan batasan yang mana yang tidak sesuai dengan batasan yang lain serta lokasi kode program yang menghasilkan batasan tersebut untuk penanganan selanjutnya.

II.2 Pohon Merah-Hitam

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 dituliskan 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).

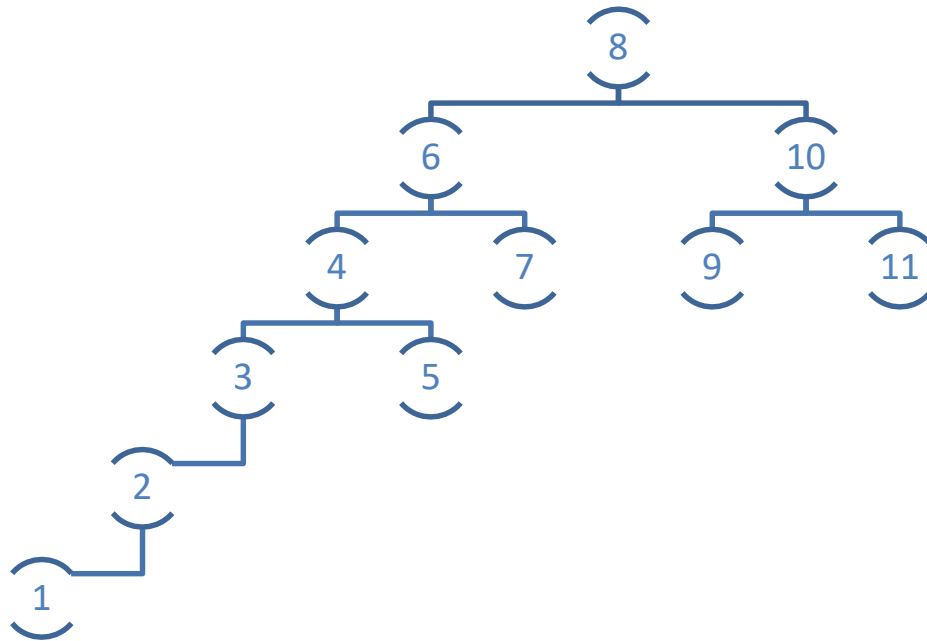
II.2.1 Objek-Objek Teori Graf

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.2.2 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 Gambar II.5, program yang akan mencari simpul “1” akan membutuhkan waktu yang lebih lama daripada program yang akan mencari simpul “11”.



Gambar II.5. 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.2.3 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:

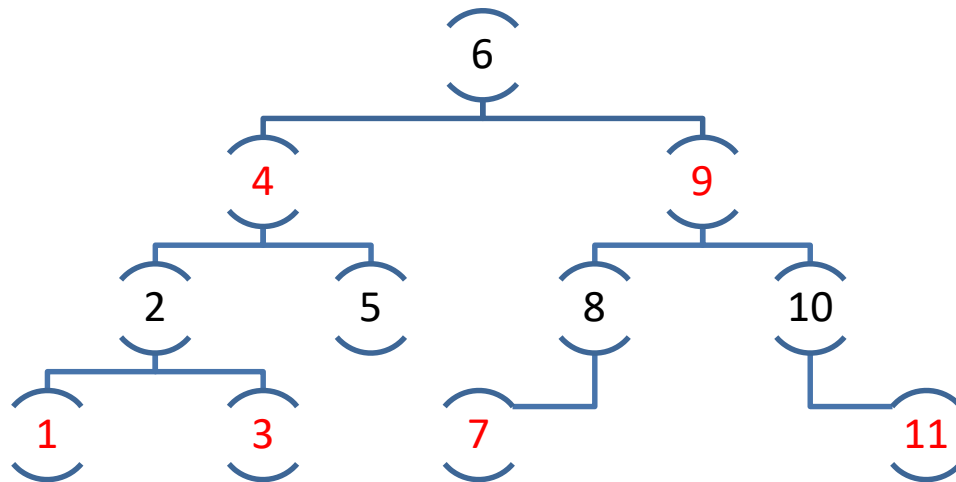
1. Properti BST; seluruh nilai pada simpul kiri dan anak-anaknya harus lebih kecil daripada nilai pada simpul ayah serta seluruh nilai pada simpul kanan dan anak-anaknya harus lebih besar daripada nilai pada simpul ayah.
2. Properti Warna; Simpul anak dari simpul berwarna merah harus memiliki warna hitam.

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, Gambar II.6 menunjukkan Gambar II.5 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.



Gambar II.6. Gambar II.5 yang diubah menjadi RBT.

II.2.4 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 sebuah 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)
```

```

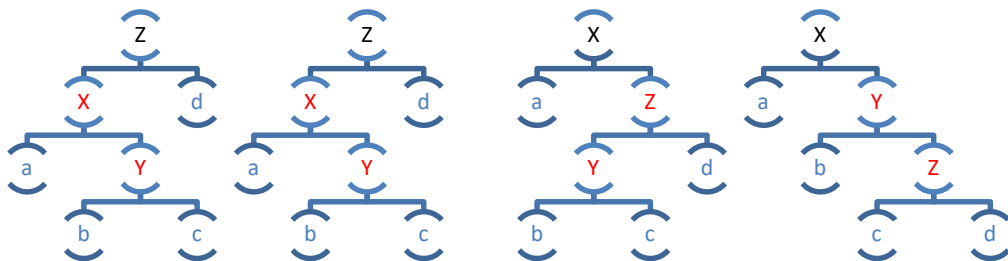
where
  ins E = T R E x E
  ins (T color a y b)
    | x < y = balance color (ins a) y b
    | x == y = T color a y b
    | x > y = balance color a y (ins b)

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

```

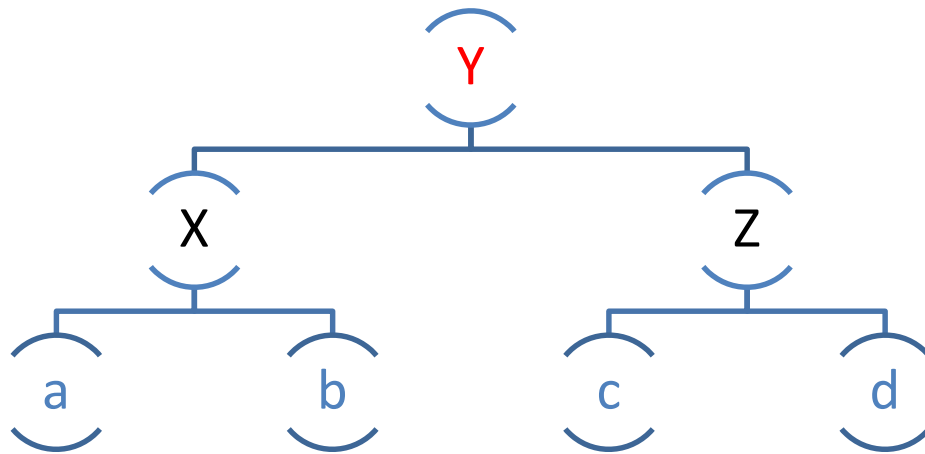
Modifikasi dari fungsi penambahan BST salah satunya adalah penambahan fungsi `makeBlack` untuk membuat simpul akar selalu berwarna hitam. Juga dipastikan bahwa simpul daun yang akan ditambahkan akan memiliki warna merah. Hal ini dilakukan untuk memastikan bahwa properti tinggi pasti akan tetap terpenuhi 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 Gambar II.7. Kasus-kasus ini mungkin terjadi ketika sebuah simpul daun berwarna merah ditambahkan sebagai simpul anak pada sebuah simpul berwarna merah.



Gambar II.7. Empat kasus pelanggaran properti warna

Algoritma Okasaki mengubah seluruh pohon ini menjadi sebuah pohon yang berbentuk seperti dapat dilihat pada Gambar II.8. Pohon ini akan memenuhi seluruh properti RBT baik properti BST, properti tinggi, maupun properti warna (Okasaki, 1999).



Gambar II.8. 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.2.5 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:

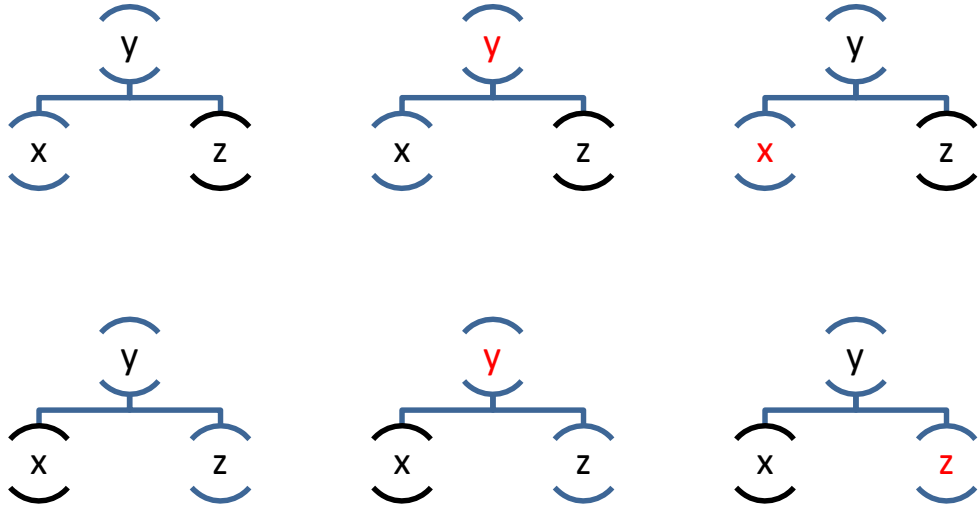
```
remove :: RBSet a -> RBSet a
-- ; Penghapusan daun
remove (T R E _ E) = E
remove (T B E _ E) = EE
-- ; Penghapusan simpul dengan satu anak
remove (T B E _ (T R a x b)) = T B a x b
remove (T B (T R a x b) _ E) = T B a x b
-- ; Penghapusan simpul dengan dua anak
remove (T color l y r) = bubble color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

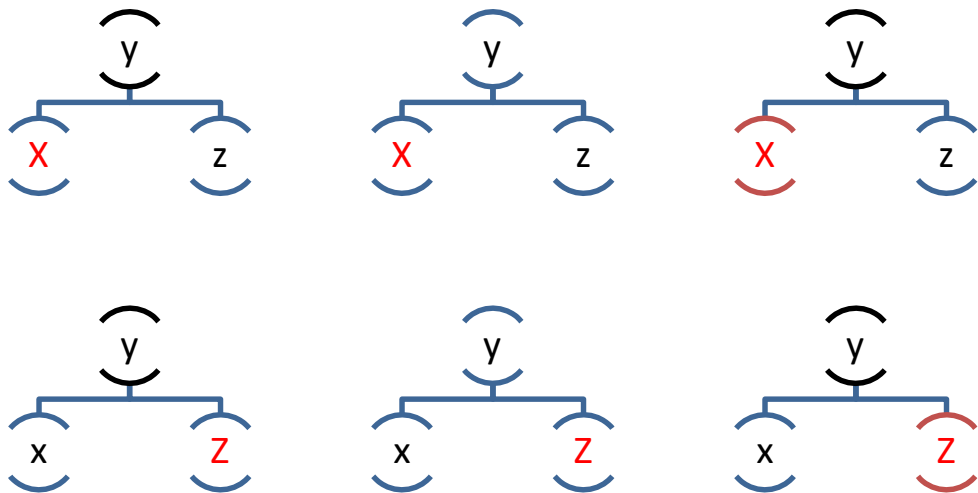
Operasi ini akan meninggalkan sebuah simpul berwarna bukan merah dan bukan hitam dalam pohon. Karena itu, maka pohon ini masih harus dimodifikasi untuk menghilangkan warna BB tersebut. Langkah pertama untuk menghilangkan warna ini adalah dengan melakukan operasi bernama *Bubble* (gelembung) yang berusaha 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 Gambar II.9.



Gambar II.9. Enam pohon yang memiliki simpul berwarna BB

Fungsi *bubble* akan mengangkat atau menghilangkan warna BB pada pohon tersebut sehingga menjadi seperti terlihat pada Gambar II.10.



Gambar II.10. Hasil operasi fungsi *bubble*

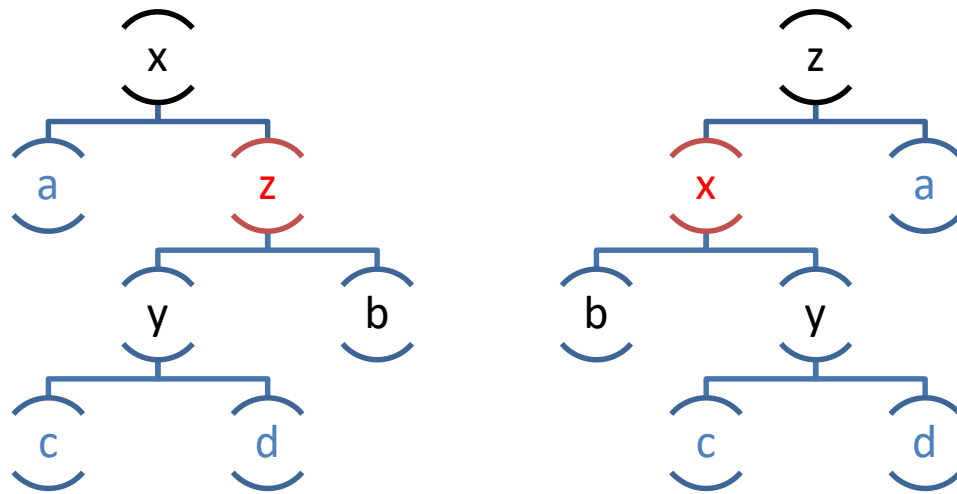
Simpul X dan Z menandakan simpul yang mungkin memiliki pelanggaran properti warna sehingga harus dijalankan fungsi penyeimbang khusus untuk simpul-simpul

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
```

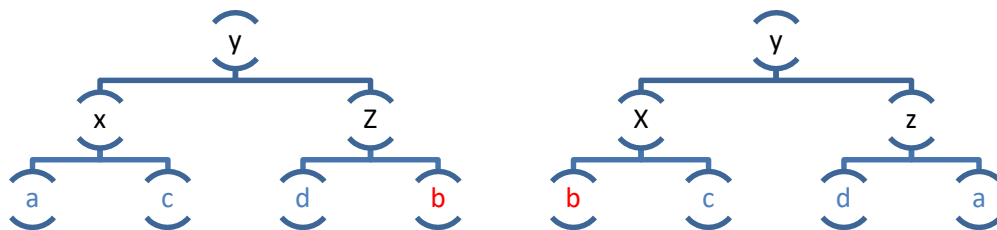
Fungsi *isBB* adalah fungsi yang mendeteksi apakah suatu simpul memiliki warna BB atau tidak. Jika tidak ada simpul anak yang memiliki warna BB, maka fungsi *bubble* bisa langsung dilewati dan program akan melakukan prosedur penyeimbangan.

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 Gambar II.7. 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 Gambar II.7, namun dengan simpul akar berwarna BB. Untuk kasus tersebut, maka pohon tersebut akan diseimbangkan sehingga menjadi pohon-pohon pada Gambar II.8, 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 Gambar II.11 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.



Gambar II.11. Pohon dengan simpul berwarna BB dan NB

Pohon-pohon ini akan diseimbangkan menjadi dua pohon yang serupa seperti dapat dilihat pada Gambar II.12. 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.



Gambar II.12. 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.3 Implementasi Pohon Merah-Hitam dan Verifikasi

II.3.1 Implementasi RBT Might dalam Bahasa Haskell

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.

II.3.2 Usaha Verifikasi Sebelumnya terhadap Program

Implementasi RBT dalam bahasa Haskell oleh Might sudah memiliki paling sedikit 2 usaha verifikasi terhadap kode tersebut. Beberapa metode yang digunakan di antaranya adalah dengan menggunakan QuickCheck dan dengan menggunakan GADT.

II.3.2.1 Verifikasi QuickCheck

Verifikasi QuickCheck ini dilakukan oleh Might sendiri terhadap programnya. Detail verifikasi ini dituliskan oleh Might dalam artikel terpisah (Might, 2014). QuickCheck bekerja dengan membandingkan sebuah struktur data yang memiliki elemen yang diacak (*randomized*) dengan fungsi tertentu yang menghasilkan nilai *True* atau *False*. Kemudian QuickCheck akan menghitung jumlah nilai *True* terhadap nilai *False* dan memberikan hasilnya dalam bentuk persentase. Program

yang memenuhi semua properti diharapkan akan mendapatkan nilai tidak kurang dari 100%. Setiap fungsi pembanding tersebut adalah representasi dari properti yang ingin dicek terhadap program.

Beberapa properti yang diverifikasi dalam eksperimen ini adalah:

- a. Pohon harus memiliki elemen yang terurut (`prop_BST`).
- b. Pohon harus memiliki simpul akar yang berwarna hitam setelah setiap operasi (`prop_Rb2`)
- c. Banyak simpul berwarna hitam untuk setiap jalur dari simpul akar ke daun harus berjumlah sama (`prop_Rb3`). Dengan kata lain, properti ini mengecek Properti Tinggi yang sudah disebutkan pada II.2.3.
- d. Setiap simpul merah harus memiliki simpul anak berwarna hitam (`prop_Rb4`). Properti ini disebut juga Properti Warna.
- e. Jika sebuah elemen dihapus dari pohon, maka pohon hasil operasi tidak boleh memiliki elemen tersebut (`prop_delete_spec1`).
- f. Seluruh elemen selain dari elemen yang dihapus harus tetap berada dalam pohon setelah operasi penghapusan (`prop_delete_spec2`).
- g. Jika elemen yang akan dihapus tidak ada dalam pohon, maka pohon yang dihasilkan harus sama dengan pohon yang menjadi masukan (`prop_delete_spec3`).
- h. Pohon hasil operasi penghapusan tetap mematuhi `prop_BST` atau dengan kata lain elemennya tetap terurut (`prop_delete_BST`).
- i. Pohon hasil operasi penghapusan tetap mematuhi `prop_Rb2`, `prop_Rb3`, dan `prop_Rb4` (`prop_delete2`, `prop_delete3`, `prop_delete4`).

Menurut verifikasi yang dilakukan oleh Might dan bisa dilakukan oleh siapapun yang memiliki program yang berada dalam artikel tersebut serta memiliki QuickCheck yang sudah terinstalasi, program ini berhasil lolos dari seluruh tes yang diberikan.

Salah satu kelebihan penggunaan verifikasi QuickCheck adalah kecepatan dan kemudahan dalam menggunakannya. Seseorang yang ingin melakukan verifikasi terhadap sebuah struktur data cukup menyediakan properti yang ingin dicek dalam

bentuk sebuah fungsi yang menerima masukan struktur data tersebut dan kemudian memberikan keluaran berupa *Boolean* atau *Property*, sebuah tipe yang khusus disediakan oleh QuickCheck untuk verifikasi lebih lanjut. Pengguna juga harus menyediakan sebuah metode khusus untuk melakukan pengacakan terhadap struktur data untuk menjadi sampel yang baik untuk verifikasi.

Namun karena sifat QuickCheck yang bergantung pada sistem pengacakan tersebut, masih ada kemungkinan bahwa QuickCheck tidak mendeteksi kesalahan karena sampel struktur data hasil pengecekan tidak representatif dan kehilangan sebuah *outlier* yang ternyata tidak memenuhi properti yang dites. Namun untuk dibandingkan dengan waktu dan biaya yang sangat murah yang harus dilakukan untuk melakukan verifikasi ini, maka seseorang yang ingin melakukan verifikasi mungkin bisa mempertimbangkan untuk menggunakan metode ini terlebih dahulu untuk menangkap kesalahan-kesalahan yang umum dan jelas sebelum melakukan verifikasi selanjutnya yang lebih menyeluruh.

II.3.2.2 Verifikasi GADT

Pada 2014 terinspirasi oleh struktur data yang sudah dibuat oleh Might, Weirich menulis ulang program RBT tersebut namun dengan menggunakan sebuah fitur Haskell bernama *Generalized Algebraic Data Type* (GADT) untuk melakukan verifikasi tepat saat penulisan program (Weirich, 2014). Fitur ini memungkinkan pemrogram untuk menuliskan struktur data dengan cara tertentu sehingga jika pengguna lain ingin menggunakan struktur data tersebut dengan cara yang salah, maka program yang ditulis tidak akan bisa berjalan, bahkan tidak akan bisa dikompilasi. Fitur ini biasanya digunakan untuk aplikasi *domain-specific* yang mungkin memiliki konvensi penulisan yang unik sehingga pengguna tidak boleh menuliskan aplikasi struktur data pada aplikasi tersebut yang tidak sesuai dengan konvensi yang sudah ditetapkan.

Implementasi RBT secara GADT milik Weirich (Weirich, 2014) memiliki empat tipe pohon. Keempat tipe tersebut adalah:

1. *RBSet*; Tipe ini menandakan RBT yang valid

2. CT (*Constructed Tree*); Tipe ini menandakan RBT yang melanggar properti yang mengharuskan simpul akar memiliki warna hitam.
3. IR (*Intermediate*); Tipe ini menandakan RBT yang memiliki simpul akar yang mungkin melanggar properti warna.
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,

Dengan pembatasan yang dilakukan oleh tipe ini, maka jika ada sebuah fungsi yang memiliki *signature* `RBSet -> RBSet`, maka fungsi itu pasti akan menerima dan menghasilkan sebuah RBT yang valid karena jika tidak, maka fungsi tersebut akan gagal untuk dikompilasi. Begitu juga untuk tipe yang lain seperti CT, IR, dan DT atau beberapa tipe lain yang digunakan sebagai pembantu dalam penulisan tipe-tipe tersebut. Tipe-tipe tersebut di antaranya adalah Valid untuk mengkodifikasi properti warna dan Incr yang menggunakan *Peano Arithmetic* untuk merepresentasikan jumlah simpul hitam untuk Properti Tinggi.

Karena program RBT yang dihasilkan oleh Weiritch dapat dikompilasi menjadi sebuah program yang berjalan, maka dapat disimpulkan bahwa program tersebut sudah berhasil diverifikasi. Usaha verifikasi ini bisa dilakukan sendiri oleh setiap orang dengan mengunduh program tersebut pada *link* yang sudah disediakan sebelumnya dan kemudian mengompilasi program tersebut dengan kompilator Haskell yang sudah dimiliki.

Verifikasi metode ini menjamin bahwa setiap program yang dihasilkan pasti mematuhi properti yang sudah ditetapkan. Namun, verifikasi dengan metode ini mengharuskan bahwa pengguna menulis ulang program dari awal dan mengharuskan pemahaman *syntax* Haskell yang lebih dalam. Struktur data yang dihasilkan oleh verifikasi ini juga tidak mudah digunakan. Oleh karena itu, verifikasi metode ini bisa direkomendasikan untuk fungsi kritis yang tidak mengizinkan kesalahan sedikit pun dalam penulisan program serta cukup berharga

untuk mendapatkan perhatian khusus dan verifikasi yang memakan lebih banyak waktu.

II.3.2.3 Spesifikasi RBT dalam makalah Liquid Haskell

Dalam salah satu makalah awal yang dituliskan mengenai Liquid Haskell, *LiquidHaskell: Experience with refinement types in the real world*, (Vazou et al., 2014) salah satu contoh permasalahan yang bisa diverifikasi oleh Liquid Haskell adalah Pohon Merah-Hitam atau RBT. Dalam makalah itu dijelaskan bahwa RBT merupakan salah satu studi kasus yang baik untuk menjadi contoh perlakuan verifikasi karena struktur data ini memiliki spesifikasi yang jelas. Spesifikasi tersebut adalah Properti BST, Properti Warna, dan Properti Tinggi yang sudah disebutkan sebelumnya. Dalam makalah itu juga tertulis bagaimana menuliskan properti-properti tersebut dalam *syntax* spesifikasi Liquid Haskell. Secara teori, verifikasi seluruh implementasi RBT dapat menggunakan spesifikasi yang sudah tertulis dalam makalah tersebut untuk langsung memverifikasi ketiga properti tersebut.

Namun dalam kasus ini, spesifikasi yang sudah tertulis dalam makalah tersebut tidak cukup untuk melakukan verifikasi implementasi RBT yang dipelajari dalam tugas akhir ini karena:

- a. Implementasi RBT Might menggunakan warna baru yaitu BB dan NB yang membuat spesifikasi yang tertulis dalam makalah harus diubah untuk memenuhi modifikasi tersebut. Ada pula perubahan urutan parameter yang tidak bisa dipahami oleh *syntax* Liquid Haskell sehingga implementasi properti RBT harus ditulis ulang dari awal.
- b. Spesifikasi tersebut hanya ditulis untuk fungsi pokok yaitu fungsi penambahan dan penghapusan. Jika fungsi tersebut menggunakan fungsi lain maka harus dituliskan spesifikasi tambahan untuk fungsi-fungsi tersebut seperti sudah dijelaskan pada II.1.1.1.2. Fungsi-fungsi baru tersebut juga mungkin menggunakan fungsi-fungsi lain yang juga membutuhkan penulisan spesifikasi lain. Pada akhirnya, spesifikasi yang

sudah tertulis pada makalah tersebut menjadi hanya sebagian kecil spesifikasi yang harus dituliskan untuk memverifikasi bahkan hanya fungsi penambahan dan penghapusan dalam implementasi RBT Might.

Oleh karena itu, walaupun solusi untuk penulisan spesifikasi RBT sudah dijelaskan dalam makalah yang dituliskan oleh Vazou, tetap dibutuhkan usaha lebih lanjut untuk mengadaptasi solusi tersebut dalam implementasi program tertentu. Namun, spesifikasi yang sudah tertulis bisa menjadi dasar dan batu loncatan untuk memudahkan penulisan spesifikasi untuk implementasi program yang dimiliki.