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

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 `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` 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 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.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 `operasiA x y = x * 2 / (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 @-}`.

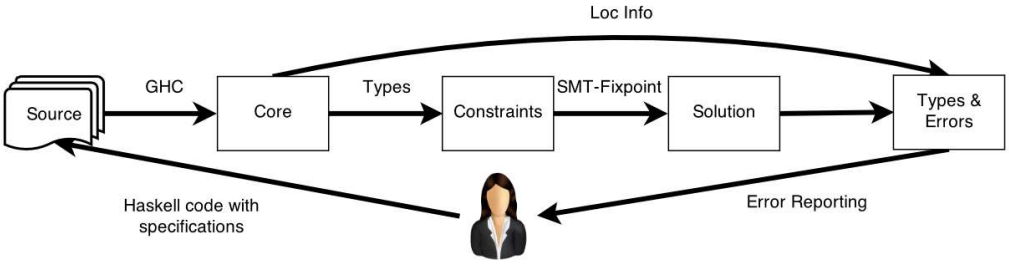
Jika fungsi `operasiA` kemudian dibutuhkan oleh fungsi lain yang mensyaratkan properti tertentu, maka spesifikasi ini bisa diubah sehingga spesifikasi ini akan mensyaratkan keluaran yang dibutuhkan tersebut. Tentu saja, fungsi ini harus tetap konsisten dan lulus dalam pengecekan konsistensi terhadap 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 $\boxed{\Gamma \vdash_{\mathrm{Q}} e : S}$

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

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

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

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

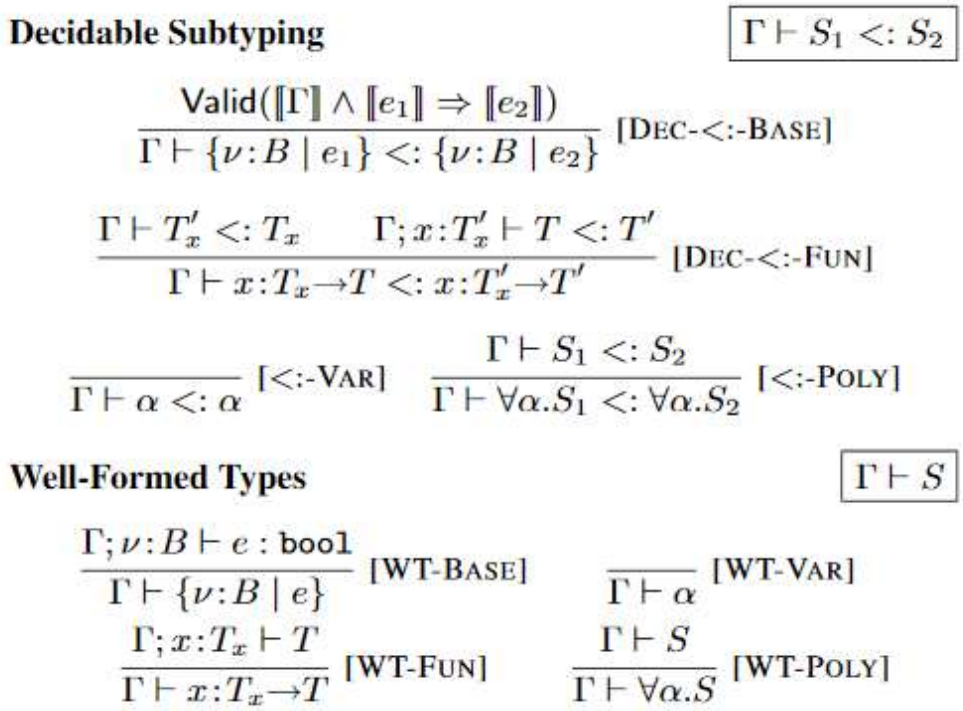
$$\frac{\Gamma \vdash_{\mathrm{Q}} e_1 : (x{:}T_x \to T) \qquad \Gamma \vdash_{\mathrm{Q}} e_2 : T_x}{\Gamma \vdash_{\mathrm{Q}} e_1\ e_2 : [e_2/x]T} \text{ [LT-APP]}$$

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

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

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

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



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 ← (p ∧ q ∧ ... ∧ 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{array}{l}
\mathsf{Cons}(\Gamma, e) = \\
\quad \textbf{match } e \textbf{ with} \\
\quad \mid\ x \longrightarrow \\
\qquad \textbf{if } \mathsf{HM}(\mathsf{Shape}(\Gamma), e) = B \\
\qquad \textbf{then } (\{\nu : B \mid \nu = x\}, \emptyset) \\
\qquad \textbf{else } (\Gamma(x), \emptyset) \\
\quad \mid\ \mathtt{c} \longrightarrow \\
\qquad (ty(\mathtt{c}), \emptyset) \\
\quad \mid\ e_1\ e_2 \longrightarrow \\
\qquad \textbf{let } (x : F_x \to F, C_1) = \mathsf{Cons}(\Gamma, e_1) \textbf{ in} \\
\qquad \textbf{let } (F'_x, C_2) = \mathsf{Cons}(\Gamma, e_2) \textbf{ in} \\
\qquad ([e_2/x]F, C_1 \cup C_2 \cup \{\Gamma \vdash F'_x <: F_x\}) \\
\quad \mid\ \lambda x.e \longrightarrow \\
\qquad \textbf{let } (x : F_x \to F) = \mathsf{Fresh}(\mathsf{HM}(\mathsf{Shape}(\Gamma), \lambda x.e)) \textbf{ in} \\
\qquad \textbf{let } (F', C) = \mathsf{Cons}(\Gamma; x : F_x, e) \textbf{ in} \\
\qquad (x : F_x \to F, C \cup \{\Gamma \vdash x : F_x \to F\} \cup \{\Gamma; x : F_x \vdash F' <: F\}) \\
\quad \mid\ \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \longrightarrow \\
\qquad \textbf{let } F = \mathsf{Fresh}(\mathsf{HM}(\mathsf{Shape}(\Gamma), e)) \textbf{ in} \\
\qquad \textbf{let } (\_, C_1) = \mathsf{Cons}(\Gamma, e_1) \textbf{ in} \\
\qquad \textbf{let } (F_2, C_2) = \mathsf{Cons}(\Gamma; e_1, e_2) \textbf{ in} \\
\qquad \textbf{let } (F_3, C_3) = \mathsf{Cons}(\Gamma; \neg e_1, e_3) \textbf{ in} \\
\qquad (F, C_1 \cup C_2 \cup C_3 \cup \{\Gamma \vdash F\} \cup \\
\qquad\quad \{\Gamma; e_1 \vdash F_2 <: F\} \cup \{\Gamma; \neg e_1 \vdash F_3 <: F\}) \\
\quad \mid\ \mathtt{let}\ x\ =\ e_1\ \mathtt{in}\ e_2 \longrightarrow \\
\qquad \textbf{let } F = \mathsf{Fresh}(\mathsf{HM}(\mathsf{Shape}(\Gamma), e)) \textbf{ in} \\
\qquad \textbf{let } (F_1, C_1) = \mathsf{Cons}(\Gamma, e_1) \textbf{ in} \\
\qquad \textbf{let } (F_2, C_2) = \mathsf{Cons}(\Gamma; x : F_1, e_2) \textbf{ in} \\
\qquad (F, C_1 \cup C_2 \cup \{\Gamma \vdash F\} \cup \{\Gamma; x : F_1 \vdash F_2 <: F\})
\end{array}
$$

```
  | [Λα]e ⟶
      let (F, C) = Cons(Γ, e) in
      (∀α.F, C)
  | [τ]e ⟶
      let F = Fresh(τ) in
      let (∀α.F′, C) = Cons(Γ, e) in
      ([F/α]F′, C ∪ {Γ ⊢ F})

Weaken(c, A) =
  match c with
  | Γ ⊢ {ν:B | θ · κ} ⟶
      A[κ ↦ {q | q ∈ A(κ) and Shape(Γ); ν:B ⊢ θ · q : bool}]
  | Γ ⊢ {ν:B | ρ} <: {ν:B | θ · κ} ⟶
      A[κ ↦ {q | q ∈ A(κ) and [[A(Γ)]] ∧ [[A(ρ)]] ⇒ [[θ · q]]}]
  | _ ⟶ Failure

Solve(C, A) =
  if exists c ∈ C such that A(c) is not valid
  then Solve(C, Weaken(c, A)) else A

Infer(Γ, e, ℚ) =
  let (F, C) = Cons(Γ, e) in
  let A = Solve(Split(C), λκ.Inst(Γ, e, ℚ)) in
  A(F)
```

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).

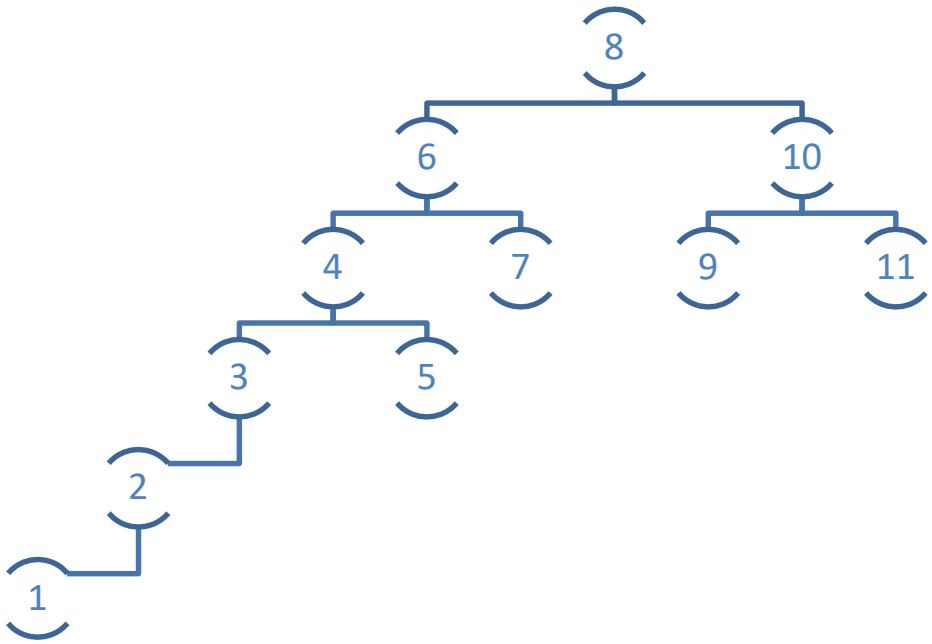
II.2.1 Objek-Objek Teori Graf

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:

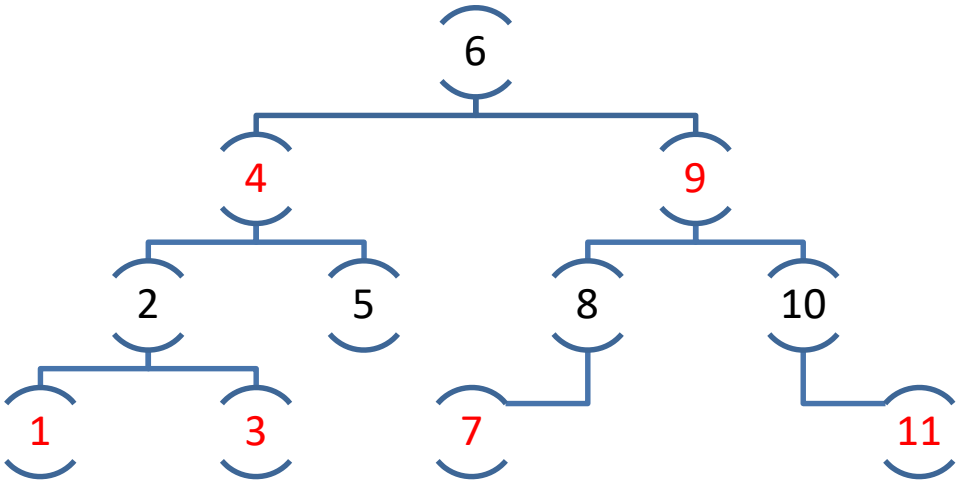
1. Properti BST; seluruh nilai pada simpul kiri dan anak-anaknya harus lebih kecil daripada nilai pada simpul ayah serta seluruh nilai pada simpul kanan dan anak-anaknya harus lebih besar daripada nilai pada simpul ayah.
2. Properti Warna; Simpul anak dari simpul berwarna merah harus memiliki warna hitam.

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 sebuat fungsi penyeimbang (*balance*) untuk menjaga properti warna RBT. Algoritma penambahan Okasaki terlihat seperti ini:

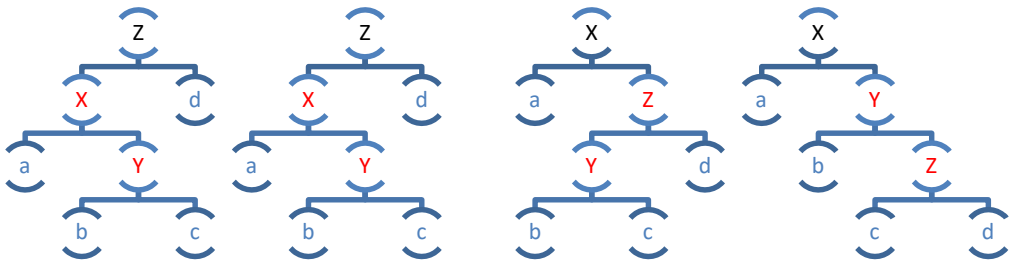
```
insert :: Ord elt => elt -> Set elt -> Set elt
insert x s = makeBlack (ins s)
```

```
    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
```

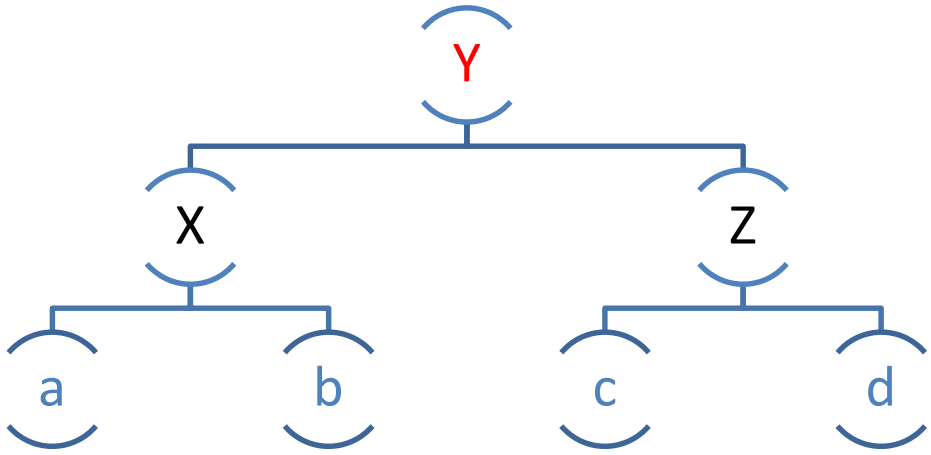
Modifikasi dari fungsi penambahan BST salah satunya adalah penambahan fungsi `makeBlack` untuk membuat simpul akar selalu berwarna hitam. Juga dipastikan bahwa simpul daun yang akan ditambahkan akan memiliki warna merah. Hal ini dilakukan untuk memastikan bahwa properti tinggi pasti akan tetap terpenuhi 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 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:

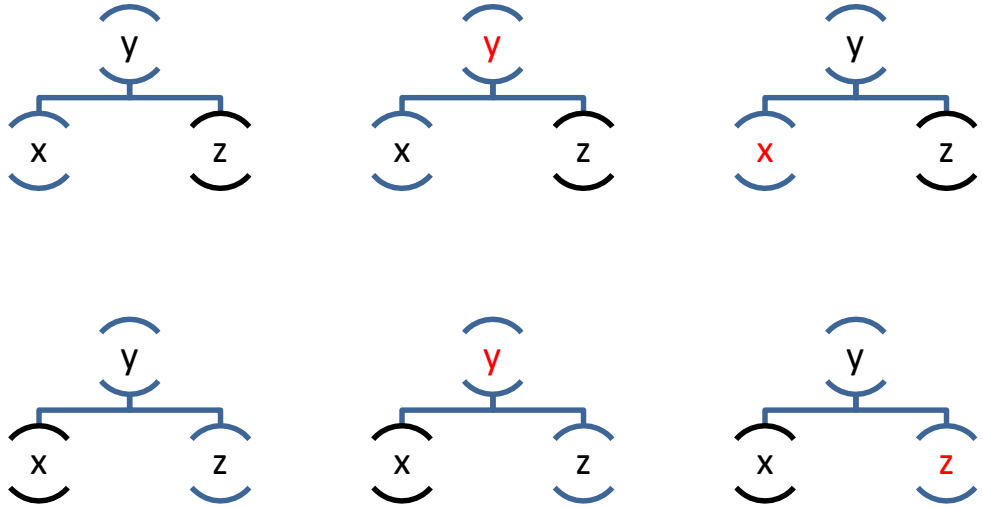
```
remove :: RBSet a -> RBSet a
-- ; Penghapusan daun
remove (T R E _ E) = E
remove (T B E _ E) = EE
-- ; Penghapusan simpul dengan satu anak
remove (T B E _ (T R a x b)) = T B a x b
remove (T B (T R a x b) _ E) = T B a x b
-- ; Penghapusan simpul dengan dua anak
remove (T color l y r) = bubble color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

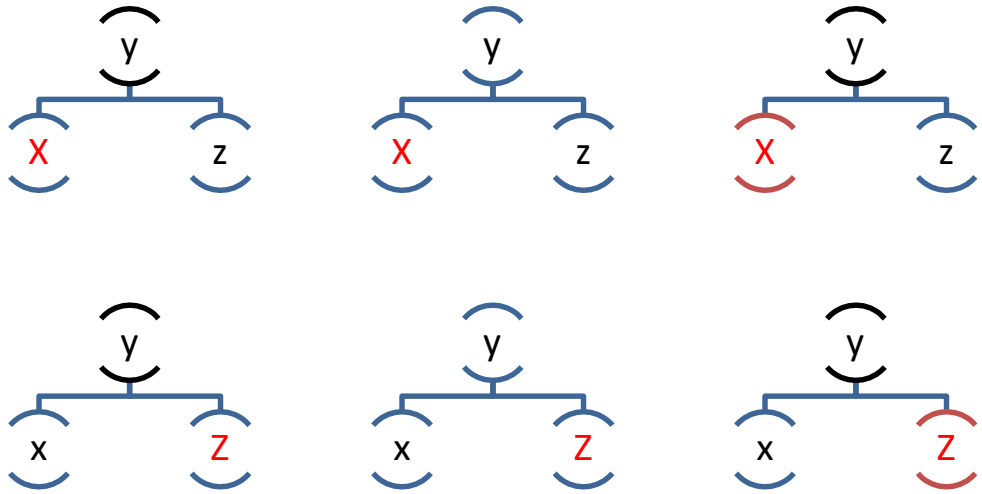
Operasi ini akan meninggalkan sebuah simpul berwarna bukan merah dan bukan hitam dalam pohon. Karena itu, maka pohon ini masih harus dimodifikasi untuk menghilangkan warna BB tersebut. Langkah pertama untuk menghilangkan warna ini adalah dengan melakukan operasi bernama *Bubble* (gelembung) yang berusaha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

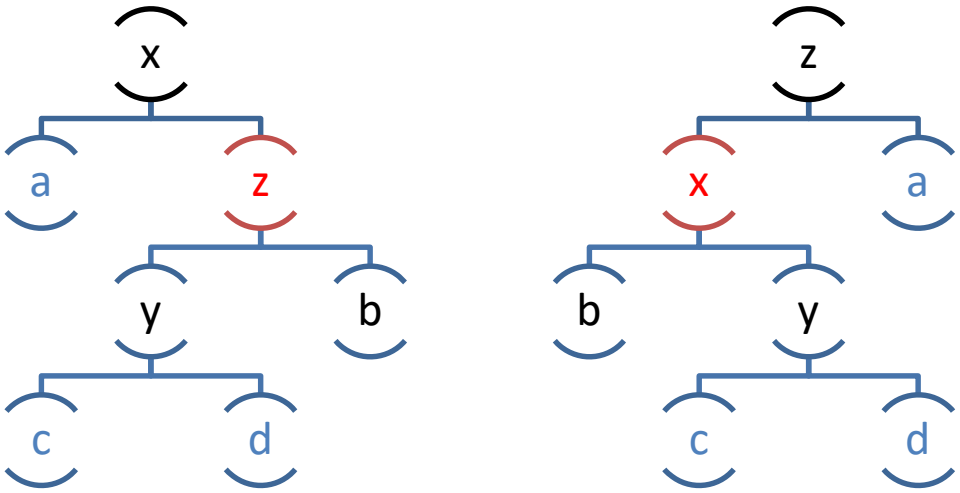
Simpul X dan Z menandakan simpul yang mungkin memiliki pelanggaran properti warna sehingga harus dijalankan fungsi penyeimbang khusus untuk simpul-simpul

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
```

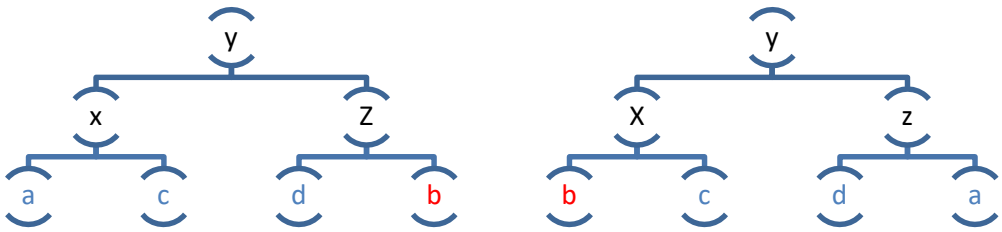
Fungsi `isBB` adalah fungsi yang mendeteksi apakah suatu simpul memiliki warna BB atau tidak. Jika tidak ada simpul anak yang memiliki warna BB, maka fungsi *bubble* bisa langsung dilewati dan program akan melakukan prosedur penyeimbangan.

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, Weiritch 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.

Pohon standar RBT yang digunakan dalam studi kasus ini memiliki definisi sebagaimana berikut:

```
data Col = R | B

data Tree a = Leaf | Node Col a (Tree a) (Tree a)
```

Untuk mengkodifikasi pohon yang memiliki akar berwarna hitam, maka fungsi pembantu `isB` memiliki *syntax* sebagaimana berikut

```
measure isB

color (Node c x l r)

color (Leaf) :: Tree a -> Prop

     = c == B

     = true
```

Fungsi ini juga digunakan untuk membantu mengkodifikasi properti warna dalam fungsi `isRB` yang berbentuk sebagaimana berikut:

```
measure isRB :: Tree a -> Prop

isRB (Leaf) = true
```

```
isRB (Node c x l r) = isRB l && isRB r &&
                      c = R => (isB l && isB r)
```

Diberikan pula spesifikasi untuk fungsi-fungsi perantara yang tidak mematuhi properti warna untuk simpul akar. Fungsi untuk merepresentasikan hal ini diberi nama `almostRB` dan berbentuk seperti berikut:

```
measure almostRB :: Tree a -> Prop

almostRB (Leaf) = true

almostRB (Node c x l r) = isRB l && isRB r
```

Kemudian untuk properti tinggi, terlebih dahulu dibuat properti tinggi untuk mencatat jumlah simpul hitam pada cabang paling kiri dalam sebuah pohon dengan *syntax* sebagaimana berikut

```
measure bh :: Tree a -> Int

bh (Leaf) = 0

bh (Node c x l r) = bh l
                    + if c = R then 0 else 1
```

Kemudian properti tinggi dapat dikodifikasi dalam fungsi `isBal` sebagaimana berikut:

```
measure isBal :: Tree a -> Prop

isBal (Leaf) = true

isBal (Node c x l r) = bh l = bh r
                       && isBH l && isBH r (sic)
```

Dalam *paper* tersebut dinyatakan bahwa meskipun fungsi bh hanya menghitung jumlah simpul hitam dalam cabang paling kiri, fungsi `isBal` tetap akan memberikan jawaban yang benar mengenai keseimbangan simpul hitam dalam sebuah pohon.

Properti BST dikodifikasi menggunakan *abstract refinement* yang memungkinkan pendefinisian struktur data yang lebih abstrak.

```
data Tree a <l::a->a->Prop, r::a->a->Prop>
     = Leaf
     | Node { c :: Col
            , key :: a
            , lt :: Tree<l,r> a<l key>
            , rt :: Tree<l,r> a<r key> }
```

Kemudian properti BST dikodifikasi dalam tipe `OTree` yang memiliki *syntax* sebagaimana berikut:

```
type OTree a = Tree <{\k v -> v<k}, {\k v -> v>k}> a
```

Pada akhirnya seluruh properti tersebut dapat digabungkan menjadi properti RBT dengan kode seperti ini:

```
type RBT a = {v:OTree a | isRB v && isBal v}
```

Untuk pohon yang digunakan oleh fungsi perantara, `isRB` bisa diganti dengan `almostRB`.

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.

BAB III

<DESKRIPSI SOLUSI>

III.1 Analisis Spesifikasi/Kebutuhan

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 karakteristik masalah yang dapat menjadi petunjuk untuk menentukan spesifikasi seperti apa yang cocok disematkan dalam sebuah fungsi dalam program.

III.1.1 Penanganan *exception*/`error`

Fungsi `error` dalam bahasa Haskell 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.1.2 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).

Dalam kasus ini, fungsi-fungsi yang sangat membutuhkan spesifikasi ini adalah fungsi-fungsi yang menghadap pengguna seperti fungsi penambahan (*insert*) dan penghapusan (*delete*). Fungsi-fungsi tersebut harus menerima pohon yang valid dan menghasilkan pohon yang valid pula. Untuk itu, spesifikasi untuk kedua fungsi ini

harus mencantumkan ketiga properti ini dalam *precondition* dan *postcondition* masing-masing.

III.1.3 Spesifikasi untuk Fungsi yang digunakan oleh Fungsi Lain

Seperti telah disebutkan dalam II.1.1.1.2, menuliskan spesifikasi untuk sebuah fungsi tidak cukup untuk melakukan verifikasi terhadap fungsi tersebut. Namun, spesifikasi juga harus dituliskan untuk seluruh fungsi yang digunakan oleh fungsi tersebut. Spesifikasi yang dituliskan tersebut tersebut harus ditulis sedemikian rupa untuk membantu verifikasi fungsi utama yang menggunakan fungsi tersebut. Dibutuhkan analisis manual untuk menentukan spesifikasi untuk fungsi-fungsi tersebut namun ada beberapa petunjuk yang dapat membantu untuk mendapatkan spesifikasi tersebut. Petunjuk salah satunya dapat dilihat pada kode GADT Weiritch yang menambahkan spesifikasi tambahan untuk seluruh fungsi pembantu tersebut. Sebagai contoh, 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.1.4 Pembuktian Kerja Fungsi Utama

Pada II.3.2.1, dapat terlihat bahwa selain spesifikasi berkaitan dengan properti RBT, Might juga memasukkan berbagai properti yang spesifik dengan bagaimana fungsi *delete* bekerja yaitu semua properti *prop_delete_*. Properti seperti ini merupakan properti yang sangat menarik untuk ditambahkan dalam spesifikasi yang harus diverifikasi ke dalam program. Namun, karena limitasi *syntax* yang disediakan oleh Liquid Haskell properti-properti ini lebih sulit dituliskan

dibandingkan properti-properti sebelumnya. Salah satu limitasi yang menyulitkan hal ini adalah cara kerja fungsi *measure* yang hanya menerima fungsi yang memiliki satu parameter. Selain itu, kurang banyak fungsi bawaan yang bisa digabungkan dalam sebuah fungsi *inline* yang bisa mengkodifikasi properti tersebut.

III.1.5 Normalisasi Seluruh Warna pada Pohon

Pada setiap operasi pada pohon RBT Might, warna-warna baru yang dikenalkan oleh Might (BB dan NB) hanya digunakan untuk sementara. Seluruh pohon yang dihasilkan oleh seluruh operasi tidak boleh memiliki warna-warna tersebut karena operasi tersebut harus menghasilkan sebuah pohon RBT normal yang bisa dikenali oleh operasi RBT konvensional yang lain. Oleh karena itu, spesifikasi untuk fungsi yang menghadap pengguna harus memastikan bahwa pohon yang dihasilkan tidak memiliki warna-warna spesial tersebut.

III.2 Spesifikasi untuk Setiap Fungsi

Berdasarkan analisis pada subbab sebelumnya, berikut adalah rencana yang lebih detail mengenai penulisan spesifikasi untuk setiap fungsi.

III.2.1 Fungsi-fungsi yang memiliki fungsi `error`

Pada sumber kode yang akan diverifikasi terdapat 4 fungsi yang menggunakan fungsi `error`. Keempat fungsi tersebut adalah fungsi `blacker`, `redder`, `max`, dan `removeMax`.

Fungsi blacker dan redder masing-masing memiliki *syntax* sebagaimana berikut:

```
blacker :: Color -> Color

blacker NB = R

blacker R = B

blacker B = BB

blacker BB = error "too black"
```

```
redder :: Color -> Color

redder NB = error "not black enough"

redder R = NB

redder B = R

redder BB = B
```

Untuk kedua kode di atas, dapat terlihat bahwa untuk menghindari `error`, spesifikasi cukup menambahkan *precondition* bahwa fungsi tidak akan menerima nilai BB untuk `blacker` dan NB untuk `redder`.

Kode untuk fungsi max dan removeMax masing-masing adalah sebagaimana berikut:

```
max :: RBSet a -> a

max E = error "no largest element"

max (T _ _ x E) = x

max (T _ _ x r) = max r


removeMax :: RBSet a -> RBSet a

removeMax E = error "no maximum to remove"

removeMax s@(T _ _ _ E) = remove s

removeMax s@(T color l x r) = bubble color l x (removeMax
r)
```

Sama seperti pada dua fungsi sebelumnya, pada dua fungsi ini spesifikasi cukup menambahkan *precondition* bahwa fungsi tidak akan menerima pohon yang kosong (E). Perlu diperhatikan bahwa fungsi-fungsi ini masing-masing merupakan fungsi rekursi atau fungsi yang menggunakan dirinya sendiri. Hal ini kadang-kadang akan menambah kompleksitas dalam penulisan spesifikasi karena fungsi itu harus

mematuhi *precondition* miliknya sendiri saat menggunakan fungsi yang merupakan dirinya sendiri. Namun untungnya dalam kasus ini, penulis program sudah menangani kode sehingga fungsi `max` dan `removeMax` tidak akan menerima pohon yang kosong karena kasus itu sudah ditangani oleh baris kode sebelumnya.

III.2.2 Penyesuaian Spesifikasi Tiga Properti RBT dalam Makalah Vazou

Seperti telah disebutkan sebelumnya dalam II.3.2.1 spesifikasi untuk tiga properti RBT telah dijabarkan dalam *paper* yang ditulis oleh Niki Vazou. Namun, dibutuhkan beberapa penyesuaian agar spesifikasi tersebut dapat digunakan dalam sumber kode ini. Sebagai permulaan, pohon RBT Might memiliki warna tambahan BB dan NB yang membuat struktur pohon tersebut menjadi agak berbeda sebagaimana berikut:

```
data Color =
       R  -- red
     | B  -- black
     | BB -- double black
     | NB -- negative black
     deriving (Show, Eq)
data RBSet a =
       E  -- black leaf
     | EE -- double black leaf
     | T Color (RBSet a) a (RBSet a)
     deriving (Show, Eq)
```

Untuk mengevaluasi properti warna, seluruh fungsi perantara menganggap bahwa pohon anak tidak memiliki warna spesial sehingga tidak terlalu banyak spesifikasi yang harus diubah. Hanya saja, karena tidak ada jaminan seperti itu untuk simpul akar, maka hamper seluruh fungsi tidak memakai fungsi `isRB` untuk mengecek pohon yang diterima melainkan menggunakan `almostRB` yang tidak mengecek

simpul akar dalam pengecekan properti warna. Dalam kasus ini, `almostRB` sudah memiliki kode yang bisa mengakomodasi warna apa pun yang dimiliki simpul akar sehingga tidak perlu ada perubahan lebih lanjut.

Untuk menghitung tinggi simpul hitam, dalam pohon RBT Might, warna spesial BB memiliki nilai 2 simpul hitam dan warna NB memiliki nilai -1. Oleh karena itu, fungsi `rb` harus dimodifikasi sedikit untuk mengakomodasi nilai-nilai baru tersebut. Fungsi `isBal` masih berfungsi sebagaimana mestinya setelah perubahan ini.

Perubahan terbesar dibutuhkan dalam kodifikasi properti BST. Dalam studi kasus dalam makalah tersebut, struktur data yang digunakan memiliki urutan parameter tertentu yang berbeda dengan urutan parameter dalam sumber kode yang akan diverifikasi. Pada makalah tersebut, parameter *key* diletakkan sebelum parameter pohon anak sedangkan pada sumber kode ini parameter *key* diletakkan setelah parameter pohon anak kiri. *Syntax* Liquid Haskell yang digunakan tidak bisa bekerja dalam jika parameter *key* diletakkan setelah sebuah parameter pohon sehingga dibutuhkan cara baru untuk mengkodifikasi properti BST untuk sumber kode ini atau sumber kode ini harus diubah agar spesifikasi dapat dikodifikasi seperti yang sudah tertulis dalam makalah tersebut.

III.2.3 Fungsi-Fungsi yang digunakan oleh Fungsi Utama

Fungsi utama `insert` dan `delete` menggunakan beberapa fungsi yang juga menggunakan beberapa fungsi lain. Fungsi `insert` memiliki *syntax* sebagaimana berikut:

```
insert :: (Ord a) => a -> RBSet a -> RBSet a
insert x s = blacken' (ins s)
 where ins E = T R E x E
       ins s@(T color a y b)
          | x < y     = balance color (ins a) y
          | x > y     = balance color a y (ins b)
```

```
    | otherwise = s
```

Dapat terlihat bawah fungsi `insert` menggunakan fungsi `blacken'` dan `balance`. Sesuai dengan penggunaannya dalam fungsi tersebut, fungsi `blacken'` dan `balance` harus memiliki *postcondition* yang sama dengan fungsi insert. Fungsi `blacken'` dan `balance` tidak menggunakan fungsi lain sehingga tidak ada fungsi lain yang membutuhkan spesifikasi tambahan.

Sementara itu fungsi `delete` lebih banyak menggunakan fungsi pembantu karena sifat fungsi tersebut lebih kompleks dari fungsi `insert`. Pada permukaannya, fungsi `delete` sendiri memiliki *syntax* yang sangat sederhana:

```
delete :: (Ord a) => a -> RBSet a -> RBSet a

delete x s = blacken (del x s)
```

Sama dengan kasus sebelumnya, `blacken` harus memiliki *postcondition* yang sama dengan `delete`. Namun, `blacken` menerima pohon dari fungsi `del` yang bisa memberikan pohon yang berbentuk apa pun. Fungsi `blacken` juga belum tentu bisa mengubah pohon berbentuk apa pun menjadi pohon yang memiliki kondisi yang sesuai dengan *postcondition* yang dimiliki oleh `delete`. Oleh karena itu, `blacken` harus menetapkan *precondition* yang seluas-luasnya yang masih memungkinkan fungsi tersebut untuk menghasilkan *postcondition* yang diinginkan dan fungsi `del` harus memiliki *postconditon* yang mematuhi *precondition* tersebut.

Salah satu petunjuk *precondition* yang dapat memenuhi hal ini terdapat dalam fungsi ekuivalen yang berada pada sumber kode GADT Weiritch. Dalam kode tersebut, fungsi `blacken` memiliki *signature*

```
blacken :: DT n a -> RBSet a
```

Tipe DT adalah tipe yang ekuivalen dengan properti `almostRB` dalam makalah Vazou. Oleh karena itu, `blacken` dapat mendapatkan *precondition* tambahan berupa `almostRB` dan `del` mendapatkan *postcondition* tambahan yang sama.

Fungsi `del` memiliki *syntax* sebagaimana berikut:

```
del x E = E
del x s@(T color a' y b')
              | x < y   = bubble color (del x a') y b’
              | x > y   = bubble color a' y (del x b')
              | otherwise = remove s
```

Fungsi `del` menggunakan fungsi `bubble` dan fungsi `remove`. Sama seperti pada fungsi `insert`, fungsi `bubble` dan fungsi `remove` juga harus memiliki *postcondition* yang sama dengan fungsi `del`.

Fungsi `remove` memiliki syntax seperti berikut:

```
remove :: RBSet a -> RBSet a
-- remove E = E   -- impossible!
-- ; Leaves are easiest to kill:
remove (T R E _ E) = E
remove (T B E _ E) = EE
-- ; Killing a node with one child;
-- ; parent or child is red:
-- remove (T R E _ child) = child
-- remove (T R child _ E) = child
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
-- ; Killing a black node with one black child:
-- remove (T B E _ child@(T B _ _ _)) = blacker' child
-- remove (T B child@(T B _ _ _) _ E) = blacker' child
-- ; Killing a node with two sub-trees:
remove (T color l y r) = bubble color l' mx r
```

```
      where mx = max l
            l' = removeMax l
```

Fungsi `remove` menggunakan fungsi `bubble`, `max`, dan `removeMax`. Di antara fungsi-fungsi ini, fungsi `bubble` sekali lagi harus memiliki *postcondition* yang sama dengan fungsi `remove`. Dalam fungsi ini, fungsi `max` dan `removeMax` sudah memiliki *precondition* yang disebutkan dalam III.2.1 oleh karena itu fungsi `remove` harus menyediakan pohon yang memenuhi *precondition* tersebut. Jika fungsi tersebut tidak mampu, dapat ditambahkan *precondition* baru pada fungsi `remove` agar hal tersebut dapat terpenuhi.

Fungsi `bubble` memiliki *syntax* seperti ini:

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

Fungsi `del` dan `remove` sama-sama menggunakan fungsi `bubble` dalam kondisi yang berbeda-beda. Fungsi `bubble` juga tidak mampu untuk menghasilkan pohon yang memenuhi *postcondition* yang diinginkan untuk setiap pohon yang ada. Oleh karena itu, seperti dengan fungsi `blacken` yang sudah disebutkan sebelumnya, fungsi `bubble` harus menetapkan *precondition* yang seluas mungkin yang masih bisa dipenuhi oleh fungsi `del` dan `remove`. Namun, tidak ada fungsi yang ekuivalen dengan `bubble` dalam kode GADT Weiritch sehingga tidak ada petunjuk yang mudah yang bisa digunakan dalam menetapkan *precondition* ini. Fungsi `bubble` juga menggunakan fungsi `blacker` dan juga fungsi `redder'` yang menggunakan fungsi `redder`. Kedua fungsi tersebut memiliki *precondition* masing-masing yang sudah disebutkan pada subbab sebelumnya. Oleh karena itu, seperti pada fungsi `remove`, fungsi `bubble` harus bisa memenuhi *precondition*

ini atau memiliki *precondition* yang bisa menyaring masukan agar *precondition* ini pasti terpenuhi.

Pada akhirnya, fungsi `balance` digunakan oleh `bubble` dan `ins`. Sama seperti fungsi `bubble`, fungsi ini juga tidak bisa memenuhi *postcondtion* yang diinginkan jika mendapatkan pohon yang berbentuk seperti apa pun. Oleh karena itu, fungsi balance harus memiliki *precondition* yang tepat yang bisa digunakan oleh kedua fungsi tersebut. Fungsi `balance` juga menggunakan dirinya sendiri (fungsi rekursif) sehingga fungsi `balance` harus mampu melayani *precondition* dan *postcondition* miliknya sendiri. Fungsi ini juga tidak memiliki fungsi yang ekuivalen dengan fungsi-fungsi dalam sumber kode GADT Weiritch sehingga dibutuhkan usaha lebih besar untuk menentukan *precondition* untuk fungsi ini.

III.3 Rencana Pelaksanaan 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.