**<CONTOH JUDUL: PENGEMBANGAN SISTEM INFORMASI DI PUSKESMAS DAERAH PEDESAAN>**

**Laporan Tugas Akhir**

**Disusun sebagai syarat kelulusan tingkat sarjana**

**Oleh**

**<NAMA MAHASISWA>**

**NIM : <NIM MAHASISWA>**



**PROGRAM STUDI TEKNIK INFORMATIKA**

**SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA**

**INSTITUT TEKNOLOGI BANDUNG**

**<Bulan> <Tahun>**

**<CONTOH JUDUL: PENGEMBANGAN SISTEM INFORMASI DI PUSKESMAS DAERAH PEDESAAN>**

**Laporan Tugas Akhir**

**Oleh**

**<NAMA MAHASISWA>**

**NIM : <NIM MAHASISWA>**

**Program Studi Teknik Informatika**

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

Telah disetujui dan disahkan sebagai Laporan Tugas Akhir

di Bandung, pada tanggal <tanggal>

Pembimbing,

<Nama dan Gelar Pembimbing>

NIP <NIP Pembimbing>

**<CONTOH JUDUL: PENGEMBANGAN SISTEM INFORMASI DI PUSKESMAS DAERAH PEDESAAN>**

**Laporan Tugas Akhir**

**Oleh**

**<NAMA MAHASISWA>**

**NIM : <NIM MAHASISWA>**

**Program Studi Teknik Informatika**

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

Telah disetujui dan disahkan sebagai Laporan Tugas Akhir

di Bandung, pada tanggal <tanggal>

|  |  |
| --- | --- |
| Pembimbing I,  <Nama dan Gelar Pembimbing I>  NIP <NIP Pembimbing I> | Pembimbing II,  <Nama dan Gelar Pembimbing II>  NIP <NIP Pembimbing II> |

**LEMBAR PERNYATAAN**

Dengan ini saya menyatakan bahwa:

1. Pengerjaan dan penulisan Laporan Tugas Akhir ini dilakukan tanpa menggunakan bantuan yang tidak dibenarkan.
2. Segala bentuk kutipan dan acuan terhadap tulisan orang lain yang digunakan di dalam penyusunan laporan tugas akhir ini telah dituliskan dengan baik dan benar.
3. Laporan Tugas Akhir ini belum pernah diajukan pada program pendidikan di perguruan tinggi mana pun.

Jika terbukti melanggar hal-hal di atas, saya bersedia dikenakan sanksi sesuai dengan Peraturan Akademik dan Kemahasiswaan Institut Teknologi Bandung bagian Penegakan Norma Akademik dan Kemahasiswaan khususnya Pasal 2.1 dan Pasal 2.2.

Bandung, <tanggal>

<Mahasiswa>

NIM <NIM>

ABSTRAK

**<CONTOH JUDUL: PENGEMBANGAN SISTEM INFORMASI DI PUSKESMAS DAERAH PEDESAAN>**

Oleh

<NAMA MAHASISWA>

NIM : <NIM MAHASISWA>

Abstrak berisi ringkasan apa yang telah dikerjakan dalam tugas akhir. Ada beberapa hal yang perlu diperhatikan dalam penulisan abstrak. Pertama, abstrak harus memuat permasalahan yang dikaji, metode/teknik yang digunakan untuk menyelesaikan masalah, hasil yang dicapai / evaluasi kajian, kesimpulan yang diperoleh, dan kata kunci. Kedua, cara penulisannya harus padat dan terarah. Setiap kalimat harus dapat memberikan informasi sebanyak dan setepat mungkin, mudah dibaca dan dimengerti. Panjang ringkasan dibatasi maksimal 300 kata dan ditulis dengan satu spasi. Panjang ringkasan dibatasi maksimal 300 kata dan ditulis dengan satu spasi.

Kata kunci: ringkasan, singkat, padat.

KATA PENGANTAR

Gunakan bagian ini untuk memberikan ucapan terima kasih kepada semua pihak yang secara langsung atau tidak langsung membantu penyelesaian tugas akhir, termasuk pemberi beasiswa jika ada. Utamakan untuk memberikan ucapan terima kasih kepada tim pembimbing tugas akhir dan staf pengajar atau pihak program studi, bahkan sebelum mengucapkan terima kasih kepada keluarga. Ucapan terima kasih sebaiknya bukan hanya menyebutkan nama orang saja, tetapi juga memberikan penjelasan bagaimana bentuk bantuan/dukungan yang diberikan. Gunakan bahasa yang baik dan sopan serta memberikan kesan yang enak untuk dibaca. Sebagai contoh: “Tidak lupa saya ucapkan terima kasih kepada teman dekat saya, Tito, yang sejak satu tahun terakhir ini selalu memberikan semangat dan mengingatkan saya apabila lengah dalam mengerjakan Tugas Akhir ini. Tito juga banyak membantu mengoreksi format dan *layout* tulisan. Apresiasi saya sampaikan kepada pemberi beasiswa, Yayasan Beasiswa, yang telah memberikan bantuan dana kuliah dan biaya hidup selama dua tahun. Bantuan dana tersebut sangat membantu saya untuk dapat lebih fokus dalam menyelesaikan pendidikan saya. ....”. Ucapan permintaan maaf karena kekurangsempurnaan hasil Tugas Akhir tidak perlu ditulis.

DAFTAR ISI

[ABSTRAK v](#_Toc66964283)

[KATA PENGANTAR vi](#_Toc66964284)

[DAFTAR ISI vii](#_Toc66964285)

[DAFTAR LAMPIRAN ix](#_Toc66964286)

[DAFTAR GAMBAR x](#_Toc66964287)

[DAFTAR TABEL xi](#_Toc66964288)

[BAB I PENDAHULUAN 1](#_Toc66964289)

[I.1 Latar Belakang 1](#_Toc66964290)

[I.2 Rumusan Masalah 1](#_Toc66964291)

[I.3 Tujuan 2](#_Toc66964292)

[I.4 Batasan Masalah 2](#_Toc66964293)

[I.5 Metodologi 2](#_Toc66964294)

[I.6 Sistematika Pembahasan 2](#_Toc66964295)

[BAB II STUDI LITERATUR 3](#_Toc66964296)

[II.1 Liquid Haskell 3](#_Toc66964297)

[II.1.1 Penggunaan Liquid Haskell 3](#_Toc66964298)

[II.1.2 Alur Kerja Liquid Haskell 8](#_Toc66964299)

[II.2 Pohon Merah-Hitam 15](#_Toc66964300)

[II.2.1 Objek-Objek Teori Graf 16](#_Toc66964301)

[II.2.2 Pohon Pencarian Biner 16](#_Toc66964302)

[II.2.3 Pohon Merah-Hitam 17](#_Toc66964303)

[II.2.4 Algoritma penambahan Okasaki 19](#_Toc66964304)

[II.2.5 Algoritma penghapusan Matt Might 21](#_Toc66964305)

[II.3 Implementasi Pohon Merah-Hitam dan Verifikasi 27](#_Toc66964306)

[II.3.1 Implementasi RBT Might dalam Bahasa Haskell 27](#_Toc66964307)

[II.3.2 Usaha Verifikasi Program Sebelumnya 28](#_Toc66964308)

[II.3.3 Modifikasi Program 30](#_Toc66964309)

[BAB III <DESKRIPSI SOLUSI> 32](#_Toc66964310)

[BAB IV <EVALUASI> 33](#_Toc66964311)

[BAB V KESIMPULAN DAN SARAN 34](#_Toc66964312)

[DAFTAR REFERENSI 35](#_Toc66964313)

DAFTAR LAMPIRAN

[Lampiran A. Contoh Judul Lampiran 36](#_Toc66964314)

[A.1 Contoh Judul Anak Lampiran 36](#_Toc66964315)

DAFTAR GAMBAR

[Gambar II.1. Sintaksis Liquid Haskell (Rondon et al., 2008) 5](#_Toc66964316)

[Gambar II.2. Liquid Haskell *Workflow* (Vazou et al., 2014) 8](#_Toc66964317)

[Gambar II.3. Aturan Pengecekan *Liquid Type* (Rondon et al., 2008) 12](#_Toc66964318)

[Gambar II.4. Algoritma *Liquid Type Inference* (Rondon et al., 2008) 14](#_Toc66964319)

[Gambar II.5. Contoh Pohon Pencarian Biner. 17](#_Toc66964320)

[Gambar II.6. Gambar II.5 yang diubah menjadi RBT. 19](#_Toc66964321)

[Gambar II.7. Empat kasus pelanggaran properti warna 20](#_Toc66964322)

[Gambar II.8. Pohon hasil penyeimbangan 21](#_Toc66964323)

[Gambar II.9. Enam pohon yang memiliki simpul berwarna BB 24](#_Toc66964324)

[Gambar II.10. Hasil operasi fungsi *bubble* 24](#_Toc66964325)

[Gambar II.11. Pohon dengan simpul berwarna BB dan NB 26](#_Toc66964326)

[Gambar II.12. Pohon hasil penyeimbangan 26](#_Toc66964327)

DAFTAR TABEL

**No table of figures entries found.**

# PENDAHULUAN

.

## Latar Belakang

Seorang pemrogram menginginkan programnya bebas dari kesalahan. Banyak waktu yang sudah terhabiskan oleh berbagai perusahaan teknologi untuk memastikan bahwa produk yang mereka rilis tidak memiliki kecacatan berarti yang mungkin saja bisa menelan korban jiwa. Kesalahan atau *bug* yang bisa ditangkap di awal proses pengembangan sebuah program bisa diatasi dengan lebih mudah daripada *bug* yang hanya disadari jauh dalam sebuah proses pengembangan.

Beberapa program untuk verifikasi membutuhkan kode untuk diverifikasi dituliskan kembali secara terpisah dalam bahasa pemrograman khusus tertentu. Pada program-program ini, pengguna harus memastikan bahwa kode yang dituliskan untuk program verifikasi merupakan program yang ekuivalen dengan program yang sebenarnya dengan usahanya sendiri. Beberapa program verifikasi lain tidak membutuhkan penulisan ulang kode program dalam bahasa terpisah yang khusus sehingga tidak perlu ada usaha tambahan untuk melakukan hal tersebut. Salah satu perangkat lunak yang dikembangkan untuk tujuan tersebut adalah Liquid Haskell untuk bahasa pemrograman Haskell. Pada dasarnya bahasa pemrograman berparadigma fungsional sudah mengalami keunggulan untuk pengaplikasian Metode Formal karena sifat bahasa tersebut yang transparan dan tidak menghasilkan “efek samping” saat mengeksekusi suatu fungsi sehingga seorang analis bisa dengan mudah melihat efek dari interaksi antara berbagai fungsi tanpa harus mempertimbangkan bahwa hasil dari interaksi tersebut mungkin berbeda dengan masukan yang sama; sama seperti sebuah fungsi matematika yang konvensional. Karena itu Haskell sebagai sebuah bahasa fungsional murni juga memungkinkan kemudahan pengaplikasian Metode Formal pada program yang dituliskan pada bahasa tersebut. Liquid Haskell memanfaatkan kemudahan ini dengan mengintegrasikan *SMT Solver* seperti Z3 atau CVC4 dan memanfaatkan perangkat tersebut untuk melakukan verifikasi pada program yang dibuat pada Haskell secara otomatis. Hal ini memungkinkan spesifikasi dituliskan langsung pada program. Kemudian dilakukan verifikasi otomatis program terhadap spesifikasi yang sudah tertulis yang membuat proses Metode Formal menjadi lauh lebih mudah dibandingkan dengan verifikasi yang dilakukan secara manual.

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

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

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

## Rumusan Masalah

Rumusan Masalah berisi masalah utama yang dibahas dalam tugas akhir. Rumusan masalah yang baik memiliki struktur sebagai berikut:

1. Penjelasan ringkas tentang kondisi/situasi yang ada sekarang terkait dengan topik utama yang dibahas Tugas Akhir.
2. Pokok persoalan dari kondisi/situasi yang ada, dapat dilihat dari kelemahan atau kekurangannya. **Bagian ini merupakan inti dari rumusan masalah**.
3. Elaborasi lebih lanjut yang menekankan pentingnya untuk menyelesaikan pokok persoalan tersebut.
4. Usulan singkat terkait dengan solusi yang ditawarkan untuk menyelesaikan persoalan.

Penting untuk diperhatikan bahwa persoalan yang dideskripsikan pada subbab ini akan dipertanggungjawabkan di bab Evaluasi apakah terselesaikan atau tidak.

## Tujuan

Tuliskan tujuan utama dan/atau tujuan detil yang akan dicapai dalam pelaksanaan tugas akhir. Fokuskan pada hasil akhir yang ingin diperoleh setelah tugas akhir diselesaikan, terkait dengan penyelesaian persoalan pada rumusan masalah. Penting untuk diperhatikan bahwa tujuan yang dideskripsikan pada subbab ini akan dipertanggungjawabkan di akhir pelaksanaan tugas akhir apakah tercapai atau tidak.

## Batasan Masalah

Tuliskan batasan-batasan yang diambil dalam pelaksanaan tugas akhir. Batasan ini dapat dihindari (tidak perlu ada) jika topik/judul tugas akhir dibuat cukup spesifik.

## Metodologi

Tuliskan semua tahapan yang akan dilalui selama pelaksanaan tugas akhir. Tahapan ini spesifik untuk menyelesaikan persoalan tugas akhir. Tahapan studi literatur tidak perlu dituliskan karena ini adalah pekerjaan yang harus Anda lakukan selama proses pelaksanaan tugas akhir.

## Sistematika Pembahasan

Subbab ini berisi penjelasan ringkas isi per bab. Penjelasan ditulis satu paragraf per bab buku.

# STUDI LITERATUR

## 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* otomatisseperti 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.

### Penggunaan Liquid Haskell

#### Penulisan Spesifikasi

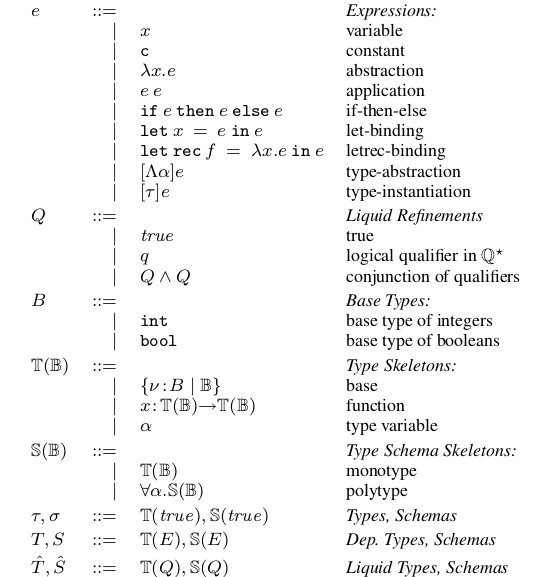
Liquid Haskell melakukan verifikasi dengan mengecek spesifikasi yang ditulis dengan bahasa khusus kemudian membandingkannya dengan fungsi yang berkaitan dengan spesifikasi tersebut. Spesifikasi dituliskan dengan menggunakan bahasa Haskell yang dimodifikasi.

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.



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

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

data Tree a =

E

| T a (Tree a) (Tree a)

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

{-@ data Tree a =

E

| T { key :: a

, lt :: Tree {v:a | v < key}

, rt :: Tree {v:a | v > key}

}

@-}

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

##### Fungsi pembantu

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

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

{-@ inline isPositiveEven @-}

isPositiveEven :: Int -> Bool

isPositiveEven x = (x > 0) && (x `mod` 2 == 0)

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

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

{-@ measure color @-}

color :: RBSet a -> Color

color (T c \_ \_ \_) = c

color E = B

color EE = BB

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

{-@ redden :: {x:RBSet a | color x == B} -> RBSet a @-}

redden :: RBSet a -> RBSet a

redden (T \_ x a b) = T R x a b

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

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

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

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

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

{-@ data Tree a =

E

| T { key :: a

, lt :: TL a key

, rt :: TR a key

}

@-}

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

##### Spesifikasi untuk Fungsi yang digunakan oleh Fungsi Lain

*Black Box. Need guarantees that functions behave in a certain way (can be anything but must fulfil this. Some functions need spec just because it’s used in other function)*

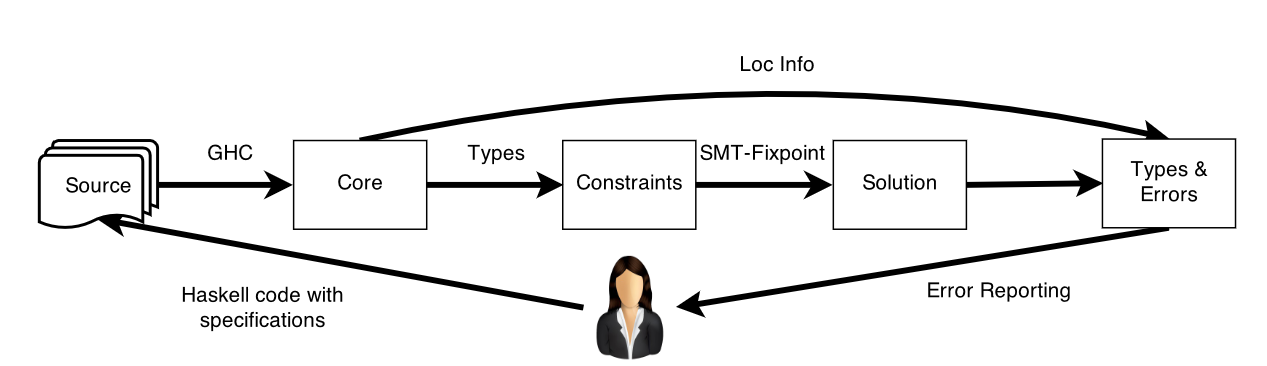
Fungsi-fungsi pada Haskell didesain untuk bisa beroperasi saling independen. Setiap fungsi bisa digunakan oleh pengguna tanpa perlu memedulikan apa pun tentang fungsi lain yang berada di *file* lain atau bahkan dalam *file* yang sama. Setiap fungsi juga dapat menggunakan fungsi lain sesuai dengan kebutuhan 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.

#### 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.

### 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)

#### 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.

#### Bahasa Perantara *Core*

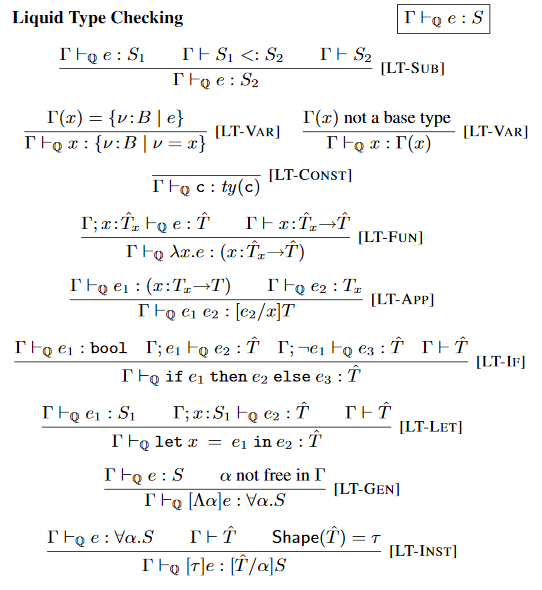
*Core* adalah sebuah bahasa perantara yang digunakan oleh GHC (*Glorious Haskell* Compiler) untuk menyatukan berbagai implementasi bahasa Haskell yang bermacam-macam menjadi sebuah bahasa yang padu dan singkat serta mudah untuk dikompilasi. Penggunaan bahasa *Core* juga memudahkan pembuatan Liquid Haskell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

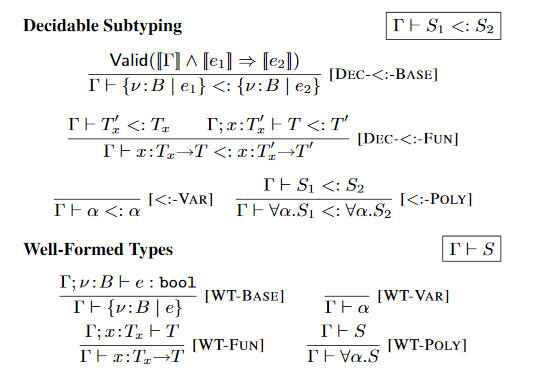
#### Kalimat Logika sebagai Batasan (*Constraints*)

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.

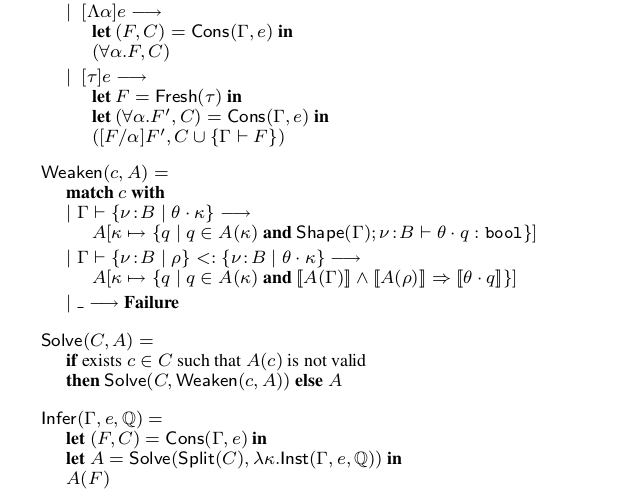
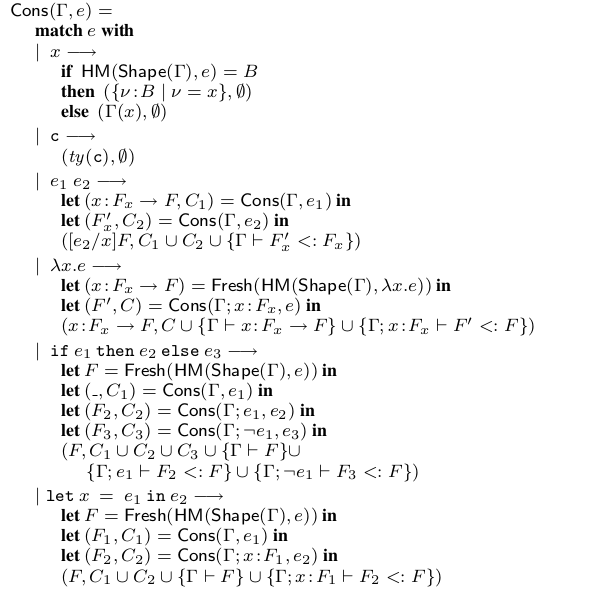




Gambar II.3. Aturan Pengecekan *Liquid Type* (Rondon et al., 2008)

#### 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.



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.

## 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 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).

### 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.

### 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.

### 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.

1. 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
2. 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.

### 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.

### 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

## Implementasi Pohon Merah-Hitam dan Verifikasi

### 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.

### Usaha Verifikasi Sebelumnya terhadap Program

Ranjit Jhala menyatakan bahwa tidak ada program yang bisa dikatakan benar (Jhala, 2018). Hanya ada program yang berhasil mematuhi spesifikasi yang sudah diberikan kepada program tersebut. Karena itu, sebelum verifikasi dilakukan terhadap sebuah program, harus ada spesifikasi yang terlebih dahulu ditentukan untuk menjadi patokan kebenaran dari implementasi sebuah program. Jika seseorang tidak memiliki kontak dengan penulis program, maka akan terjadi kesulitan untuk menentukan apakah masukan dan keluaran yang sebenarnya diinginkan oleh penulis program saat menulis suatu fungsi. Namun, ada beberapa petunjuk yang bisa dilihat untuk menentukan spesifikasi seperti apa yang cocok disematkan dalam sebuah fungsi dalam program.

#### Tiga Properti RBT

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

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

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

#### Spesifikasi Literatur Sebelumnya untuk Fungsi Perantara

Tidak semua fungsi dalam implementasi struktur data RBT selalu mematuhi ketiga properti yang sudah disebutkan sebelumnya. Ada beberapa fungsi perantara yang digunakan oleh program dan tidak bisa diakses oleh pengguna. Fungsi-fungsi ini tidak didesain untuk mematuhi ketiga properti tersebut untuk memudahkan pemrograman. Namun, pada akhirnya saat fungsi-fungsi yang memanfaatkan fungsi-fungsi perantara tersebut harus tetap mampu menghasilkan RBT yang valid.

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

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

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

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

# <DESKRIPSI SOLUSI>

## Fungsi-Fungsi dalam Sumber Kode

*Fungsi Utama (Insert, Delete), Fungsi yang dipakai dalam Fungsi Utama, Fungsi Lain*

## Penulisan Spesifikasi untuk Fungsi yang Relevan

*Bisa ambil banyak dari III.2 sebelumnya, tambahin fungsi yang relevan untuk masing-masing.*

Tujuan penulisan bab ini adalah untuk:

1. menguraikan hasil analisisyang lebih mendalam tentang persoalan tugas akhir, serta
2. menguraikan pendekatan dan detil solusi yang dipakai untuk menyelesaikan persoalan tugas akhir.

Judul dan jumlah bab Deskripsi Solusi disesuaikan dengan topik tugas akhir, yang mencerminkan pendekatan yang dilakukan untuk menyelesaikan masalah tugas akhir. Bergantung pada banyaknya materi yang akan dibahas, bab Deskripsi Solusi dapat di pecah menjadi beberapa bab sesuai dengan kebutuhan.

Bab ini, jika memungkinkan, sebaiknya didahului dengan hasil analisispersoalan tugas akhir. Tujuan utamanya adalah untuk memberikan pemahaman yang lebih baik aspek-aspek penting yang menyebabkan persoalan tugas akhir muncul. Hasil analisis tersebut difokuskan pada hal-hal yang relevan sehingga dapat mengarahkan (memberikan ide) untuk pemikiran jenis solusi yang dapat diterapkan untuk menyelesaikan persoalan tersebut. Untuk topik tugas akhir yang solusinya berupa pengembangan aplikasi, hasil analisis dapat berupa kebutuhan fungsional dan non fungsional.

Bagian terbesar dari bab ini adalah menguraikan dengan detil konsep solusi yang Anda tawarkan.

# <EVALUASI>

Tujuan penulisan bab ini adalah untuk menunjukkan seberapa jauh solusi yang diuraikan pada bagian sebelumnya dapat menyelesaikan permasalahan utama Tugas Akhir. Metode yang dipakai untuk melakukan evaluasi dapat bermacam-macam, bergantung pada jenis permasalahannya.

# KESIMPULAN DAN SARAN

Bab Kesimpulan dan Saran merupakan penutup dari bagian utama Laporan Tugas Akhir. Fokuskan kesimpulan pada hal-hal baru yang relevan dengan ketercapaian tujuan Tugas Akhir terkait dengan permasalahan yang diselesaikan dalam Tugas Akhir. Saran berisi kajian hal-hal yang masih dapat dikembangkan lebih lanjut.

DAFTAR REFERENSI

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

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

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

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

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

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

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

Rondon, P. M., Kawaguchi, M., & Jhala, R. (2008). Liquid types. *ACM SIGPLAN Notices*, *43*(6), 159–169. https://doi.org/10.1145/1379022.1375602

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

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

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

1. Contoh Judul Lampiran
   1. Contoh Judul Anak Lampiran

Contoh anak lampiran