SEBUAH *FRAMEWORK* UNTUK MEKANISASI MULTI LOGIKA

I.S.W.B. Prasetya, A. Azurat dan S.D. Swierstra

Instituut van Informatiekunde & Informatica, Universiteit Utrecht.
email : wishnu@cs.uu.nl, doaitse@cs.uu.nl, dan ade@cs.uu.nl

ABSTRAK

Otomasi dari verifikasi formal sebuah sistem membutuhkan mekanisasi logika yang menjadi basis metoda verifikasi yang digunakan. Logika yang dibutuhkan sering kali cukup rumit dan sebetulnya merupakan komposisi dari beberapa logika lain-nya. Ini memberikan komplikasi ekstra karena sekarang aspek seperti hirarki antar logika dan modularitasnya merupakan aspek yang juga perlu diperhatikan. *Framework* yang ada cenderung berfokus pada mekanisasi dari sebuah logika saja dan ini menurut pengalaman kami kurang memuaskan untuk membangun sistem dengan multi logika. Dalam tulisan ini kami memberikan sebuah *framework* alternatif yang diharapkan lebih cocok untuk keperluan tersebut.

Kata kunci : logika, verifikasi, metode formal.

1. PENDAHULUAN

Metode formal diakui memiliki potensi besar dalam pengembangan sistem yang bermisi kritis (seperti sistem pengendali lalu lintas, pesawat terbang, dan lain-lain) terutama dalam tahapan pengujian (verifikasi). Pelaksanaan metode formal pada dasarnya adalah penerapan dari logika pemrograman. Secara umum logika merupakan rumusan eksak dari sebuah strategi untuk menurunkan (*to infer*) sifat-sifat tertentu dari sebuah sistem. Dalam logika sifat-sifat tersebut dinyatakan dengan rumusan matematis. Penggunaan rumusan matematis inilah yang akan menjamin deskripsi sistem dinyatakan dengan eksak sehingga ketidaksesuaian antara kebutuhan dan sistem yang dihasilkan dapat lebih mudah dideteksi.

Kendala dalam penggunaan metoda formal adalah bahwa penggunaannya secara manual terlalu sulit. Hal ini disebabkan karena terlalu sulit bagi manusia untuk dapat menjabarkan dan menganalisa rumusan matematis yang terlalu panjang dan kompleks. Peluang terjadinya kesalahan sangat besar. Jalan keluarnya adalah otomasi. Sejauh ini, otomasi dari metoda formal masih dalam fase pengembangan. Aplikasinya masih belum meluas walaupun di beberapa bidang tertentu seperti verifikasi pemrosesan mikro, metoda ini mulai banyak digunakan. Untuk dapat melakukan otomasi metode formal, kita membutuhkan proses yang disebut *mekanisasi*, yaitu proses mengimplementasikan logika dalam sistem komputer.

Dalam literatur dapat ditemukan berbagai pendekatkan untuk *mekanisasi*. Pendekatan-pendekatan tersebut dapat dikelompokkan dalam tiga kategori:

1. Mekanisasi tunggal yaitu meng-implementasikan sebuah logika langsung di bahasa pemrograman. Sebuah contoh mekanisasi tunggal adalah dalam [1] mengimplementasikan *First Order Logic* dalam bahasa ML. Alat verifikasi model checker, juga dapat dikategorikan juga sebagai implementasi langsung, seperti UV (versi 1.20) [2] yang merupakan model checker dari logika UNITY [3] yang ditulis dalam bahasa C.
2. Embedding dalam sebuah logika lain. Disini sebuah logika dimekanisasi dengan cara merepresentasikannya dalam sebuah logika lain (selanjutnya akan disebut logika *host*) yang telah dimekanisasi. Dengan cara ini, logika yang di-*embed* secara otomatis juga akan termekanisasi. Keuntungan utama dengan metode ini adalah, bahwa implementasi logika akan dapat dapat dijamin konsistensinya terhadap logika host tersebut, dan memungkinkan kita untuk dapat melakukan penalaran terhadap logika pemrograman yang di-*embed* (fasilitas ini biasa disebut sebagai penalaran meta). Sebagai contoh, *HOL* theorem prover [4] merupakan host logic yang banyak digunakan. Tool ini merupakan mekanisasi dari *Higher Order Logic*. Contoh dari embedding adalah mekanisasi dari UNITY [5][6] di dalam HOL.
3. Mekanisasi pada logika yang generik. Logika generik merupakan logika yang minimal yang dibangun khusus sebagai logika dasar (*meta logic*). Logika dasar ini memiliki fungsi-fungsi utama yang cukup general. Mekanisasi dibangun

Makalah diterima [16 Agustus 2001]. Revisi akhir [25 November 2001]

menggunakan fungsi-fungsi tersebut. Dengan pendekatan ini penalaran meta masih dapat dilakukan sebatas tingkat ekspresif dari logika dasarnya. *Isabelle* [7] merupakan salah satu tools yang biasa disebut juga sebagai *generic thoerem prover*. *Isabelle* dilandasi dengan *intuinistic logic* sebagai logika dasar. Beberapa logika seperti *HOL (Higher Order Logic)* dan *FOL (First Order Logic)* telah diimplementasikan dalam *Isabelle*.

Pada prakteknya sebuah logika yang dipakai untuk membuktikan sifat-sifat abstrak dari sistem yang non-trivial sering kali merupakan komposisi dari beberapa sub-logika, dimana logika logika pemrograman sebenarnya hanyalah salah satu bagian dari logika akhir yang dipakai. Misalnya, kita sering masih membutuhkan logika predikat atau logika set untuk memformulasikan dan membuktikan spesifikasi dari sistem. Sistem multi logika seperti ini tidak cocok, paling tidak dari sudut pandang arsitektur software, untuk dibangun sebagai sebuah kesatuan yang monolitis. Isu dari sistem multi komponen akan kita jumpai juga disini, seperti modularitas, *code re-use*, organisasi modul, dan *interfacing* antar modul. Karena dalam sistem yang besar isu-isu multi komponen seperti ini menjadi cukup rumit, kita membutuhkan sebuah arsitektur mekanisasi yang fleksibel dimana komposisi komponen bisa dengan mudah dilakukan dan diekspresikan dengan elegan.

Ketiga pendekatan yang sudah ada, memiliki keterbatasan dalam merepresentasikan sistem multi logika.

Pendekatan kedua maupun ketiga tidak memiliki fasilitas yang *first class* untuk melakukan komposisi komponen. Fasilitas seperti ini ada, tetapi lebih merupakan fasilitas dari bahasa implementasi yang digunakan oleh logika *host*, dan penggunaannya di tingkat logika *guest* terasa kurang alamiah , dan dari sudut pandang arsitektur perangkat lunak kurang elegan.

Pada pendekatan pertama sebuah logika diimplementasikan sebagai sebuah kesatuan yang monolitis sehingga kita sulit untuk mendapatkan akses ke modul-modul dari logika tersebut yang mungkin bisa kita gunakan ulang dalam komposisi dengan logika lain. Akan tetapi dalam pendekatan ini mekanisasi dilakukan langsung dengan sebuah bahasa pemrograman. Beberapa bahasa pemrograman seperti Java atau SML memiliki fasilitas yang cukup handal untuk memprogram komposisi komponen. Ini tentunya bisa kita manfaatkan.

Karena kendala yang kita hadapi kalau menggunakan ketiga pendekatan tersebut, dalam tulisan ini kami mencoba memberikan *framework* atau konsep dari sebuah arsitektur baru yang diharapkan lebih cocok untuk melakukan mekanisasi dari sistem multi logika.

Framework yang diajukan dapat dilihat sebagai pengembangan dari pendekatan pertama, dimana ketimbang kita mengganti struktur mekanisasi yang monolitis dengan struktur yang moduler. Pada dasarnya ini juga dapat dilakukan dengan menggunakan pendekatan kedua atau ketiga, tetapi implementasi langsung (tanpa lewat embedding) tetap akan memberikan programmability yang lebih besar, terutama dalam hal ini, karena kita bisa memilih bahasa implementasi yang memang cocok untuk melakukan komposisi komponen. Java merupakan sebuah alternatif yang baik, tetapi permasalahannya adalah sintaks dari sebuah logika serta hukum-hukum inferensinya lebih mudah diimplementasikan menggunakan bahasa pemrograman fungsional [1]. Karena itu kita memilih untuk menampilkan *framework* ini dalam Haskell, yang merupakan bahasa pemrograman fungsional, tetapi juga memiliki fasilitas kelas yang cukup untuk mengekspresikan modularitas yang kita butuhkan. Logika sebagai komponen yang reusable akan direpresentasikan sebagai kelas dalam Haskell yang didalamnya implisit memiliki fasilitas penurunan sifat untuk code re-use.

2. LOGIKA DAN BAHASA

Logika digunakan orang untuk membuktikan secara formal sifat-sifat tertentu dari obyek-obyek dunia nyata. Untuk memformulasikan sifat-sifat tersebut secara eksak/persis kita membutuhkan sebuah bahasa. Bahasa yang digunakan bisa bermacam-macam dan disesuaikan dengan kebutuhkan kita. Dalam hal ini, bahasa disebut juga basis dari logika tersebut. Disamping itu sebuah logika juga terdiri dari hukum-hukum dasar yang mengatur langkah pembuktian apa saja yang diijinkan dalam logika tersebut. Hukum-hukum tersebut disebut juga hukum inferensi. Sebagai contoh, berikut ini adalah sebuah hukum dari logika predikat yang dikenal dengan nama *modus ponens* :

$$\frac{p \ , \ p \Rightarrow q}{q} \qquad (1)$$

Sebuah hukum inferensi dari sebuah logika pada dasarnya adalah sebuah fungsi yang melakukan transformasi pada kalimat-kalimat dari logika tersebut. Setiap langkah dalam sebuah pembuktian adalah aplikasi dari salah satu hukum inferensi dan

sebuah pembuktian adalah komposisi dari hukum-hukum inferensi.

Implementasi dari sebuah logika disebut juga *mekanisasi*. Tujuannya adalah supaya eksekusi dari hukum-hukum inferensi dilakukan oleh komputer sehingga aspek kesalahan manusia disini menjadi nihil (minimum). Disamping itu mekanisasi juga memberikan basis untuk otomasi dari proses pembuktian. Dalam melakukan mekanisasi dari sebuah logika ada dua hal yang harus kita implementasikan: (1) representasi dari bahasa yang menjadi basis dari logika tersebut, dan (2) hukum-hukum inferensi.

Bahasa pemrograman fungsional seperti Haskell atau ML merupakan pilihan yang baik untuk melakukan mekanisasi logika, yaitu karena gramatika sebuah bahasa dapat direpresentasikan dengan mudah menggunakan data tipe; sedangkan hukum inferensi adalah fungsi yang bekerja diatas data tipe tersebut. Konsep seperti ini lebih cocok dinyatakan sebagai *Abstract Data Type* dalam bahasa fungsional dibandingkan dalam bahasa *object oriented*. Sudah banyak contoh dari implementasi logika dengan menggunakan bahasa fungsional, misalnya HOL dalam ML, Coq (dalam CaML), Lava (Haskell).

Dalam tulisan ini kita akan menggunakan Haskell sebagai bahasa pemodelan sekaligus bahasa implementasi. Haskell adalah bahasa pemrograman fungsional. Bahasa ini cukup abstrak sehingga tidak salah bila digunakan sebagai bahasa pemodelan. Terlebih lagi, karena bahasa ini adalah bahasa pemrograman, maka model yang kita buat juga langsung merupakan implementasi.

Alasan lain untuk memilih Haskell adalah karena mekanisme kelasnya yang intuitif. Kelas di Haskell digunakan untuk mendefinisikan *interface* yang seragam untuk mengakses berbagai struktur data.

3. NOTASI

Berikut ini adalah rangkuman dari beberapa aspek bahasa dari Haskell yang mungkin agak kurang dikenal.

Apabila `a` adalah sebuah tipe, maka `[a]` adalah tipe dari list yang anggotanya adalah obyek-obyek dari tipe `a`.

Kita juga akan menggunakan konsep kelas dari Haskell [8]. Sebagai contohnya adalah kelas `Eq` berikut ini:

```
Class Eq a where

== :: a->a->Bool
/= :: a->a->Bool
```

yang artinya adalah sebagai berikut. Apabila sebuah tipe `a` adalah instansiasi dari kelas `Eq` maka kita bisa mengasumsikan bahwa fungsi-fungsi dengan nama dan tipe seperti diatas sudah tersedia. Fungsi-fungsi diatas bisa dilihat sebagai *interface* untuk mengakses data dari sebuah tipe dan kelas mendefinisikan *interface* yang seragam untuk mengakses data dari berbagai tipe, walaupun untuk tipe yang berbeda bisa jadi implementasi dari *interface*nya berbeda. Apabila kita mendefinisikan sebuah tipe, kita dapat menyatakan bahwa tipe tersebut merupakan instansiasi dari sebuah kelas *C*. Ini artinya kita memutuskan bahwa tipe tersebut mendukung *interface* yang dijanjikan *C*. Dengan kata lain, kita harus menyediakan implementasi dari *interface* tersebut. Contoh instansiasi adalah sebagai berikut:

```
instance Eq Bool where

True  == True  = True
False == False = True
_     == _     = False

x /= y = not (x == y)
```

yang mendeklarasikan tipe `Bool` sebagai instansiasi dari kelas `Eq`.

Kelas dapat juga dilihat sebagai pengelompokan dari tipe-tipe yang memiliki *interface* sama. Sebuah tipe juga dapat mendukung lebih dari satu *interface* (kelas). Konsep kelas dari Haskell ini bisa juga disamakan dengan konsep *class interface* dari Java.

4. BAHASA DAN TRANSFORMASI KALIMAT

Dalam teori bahasa sebuah bahasa didefinisikan sebagai himpunan dari *kalimat*. Untuk menspesifikasikan sebuah bahasa biasanya orang menggunakan apa yang disebut *gramatika*. Sebagai contoh berikut ini adalah sebuah gramatika G_1 dari bahasa yang digunakan dalam logika proposisi yang hanya menggunakan $\Rightarrow$, $\wedge$, dan $\neg$:

Formula $\rightarrow$
- | T | F
- | *Indentifier*
- | $\neg$ *Formula*
- | *Formula* $\wedge$ *Formula*
- | *Formula* $\Rightarrow$ *Formula*
- | (*Formula*)

Dalam bahasa-bahasa fungsional seperti Haskell atau ML, gramatika dapat direpresentasikan dengan data tipe. Sebagai contoh, gramatika diatas dapat direpresentasikan sebagai berikut:

```
data A1 =
     A1_Const Bool
   | A1_Iden  String
   | A1_Not   A1
   | A1_And   A1 A1
   | A1_Imp   A1 A1
```

Setiap value dari tipe `A1` tidak hanya merepresentasikan sebuah kalimat dari gramatika G_1, tetapi juga struktur dari kalimat tersebut. Sebagai contoh, value:

```
A1_Imp
  (A1_Iden "p")
  (A1_Const True)
```

merupakan representasi dari kalimat $p \Rightarrow$ True.

Representasi seperti `A1` sangat praktis dalam implementasi logika karena dalam logika kita banyak melakukan transformasi dan analisis kalimat. Jadi, kita akan menggunakan tipe seperti `A1` untuk merepresentasikan sebuah bahasa.

Dua operasi dasar yang dilakukan pada kalimat adalah evaluasi dan transformasi. Evaluasi adalah operasi untuk mengartikan sebuah kalimat, sedangkan transformasi adalah operasi yang mengubah bentuk sebuah kalimat menjadi kalimat lain.

Penerapan hukum inferensi sebuah logika pada dasarnya adalah transformasi kalimat. Dalam hal ini kalimat yang masuk dan keluar berasal dari bahasa yang sama, yaitu bahasa dari logika tersebut.

Contoh lain dari transformasi formula adalah *kompilator*. Sebuah kompilator dari bahasa `a` ke bahasa `b` mengubah kalimat-kalimat dari bahasa `a` menjadi kalimat-kalimat berbahasa `b`. Sementara kompilator yang umum dikenal dalam dunia programming, adalah transformasi dari sebuah bahasa pemrograman kedalam bahasa mesin (*binary code*).

5. LOGIKA DAN MEKANISASI

Sebuah logika L terdiri dari hukum-hukum inferensi. Hukum inferensi sering dispesifikasikan, sebagai contohnya, seperti ini:

$$\frac{f \;,\; g}{h} \qquad (2)$$

artinya adalah: kalau kalimat f dan g bisa dibuktikan, maka kalimat h bisa disimpulkan dari f dan g. Pada prakteknya, dalam membangun pembuktian mekanis orang lebih banyak menggunakan apa yang disebut *taktik* ketimbang menggunakan hukum inferensi secara langsung. Taktik bekerja mundur. Jadi kita mulai dari h. Ini bisa dilihat sebagai hipotesa atau tujuan pembuktian — sering juga disebut *goal*. Berdasarkan hukum diatas kita dapat mereduksi h menjadi f dan g. Kita kemudian melanjutkan proses pembuktian dengan berusaha mereduksi f dan g. (Sebuah formula dinyatakan valid bila dapat di reduksi menjadi *empty*).

Dalam contoh diatas, kalimat h diubah bentuknya menjadi dua formula lainnya. Kita akan melihat proses ini secara lebih umum lagi, yaitu dengan mendefinisikan bahwa taktik bekerja pada list dari dari kalimat ketimbang pada sebuah kalimat saja.. Jadi, apabila `a` adalah bahasa dari logika L, maka taktik adalah fungsi dengan tipe `[a] -> [a]`. Kita akan menambahkan dua aspek teknis lain:

1. Sebuah taktik dapat gagal. Ini akan diindikasikan dengan value `Fail`.
2. Adakalanya sebuah taktik melakukan transformasi ke bahasa lain.

Jadi, model yang akan kita gunakan adalah sebagai berikut:

```
data Try a  = Ok a | Fail

type Goal a = Try [a]

Tactic a b = Goal a -> Goal b
```

Sebuah logika berbahasa `a` pada dasarnya dapat dilihat sebagai himpunan dari taktik-taktik. Cara seperti ini dalam merepresentasikan sebuah logika memiliki keuntungan bahwa apabila kita memformulasikannya dalam bahasa pemrograman fungsional maka ia berfungsi juga sebagai implementasi dari logika tersebut. Sebagai contoh, kita lihat logika L_1 yang berbasis bahasa `A1`. Logika ini adalah 'subset' dari logika proposisi yang sangat sederhana. Ia hanya memiliki dua taktik, yaitu:

$$\frac{p \wedge q}{p \;,\; q}\ \texttt{conj\_tac} \qquad (3)$$

$$\frac{p \Rightarrow q,\ q}{p,\ p \Rightarrow q}\ \texttt{mp\_tac} \qquad (4)$$

Dalam ML, kedua taktik ini dapat kita implementasikan sebagai berikut:

```
L1_conj_tac :: Tactic A1

L1_conj_tac (Ok [A1_And p q])
  =
  (Ok [p,q])

L1_conj_tac _ = Fail


L1_mp_tac :: Tactic A1

L1_mp_tac (Ok [A1_Imp p q, q'])
  =
  if q==q'
  then Ok [p, A1_Imp p q]
  else Fail

L1_mp_tac _ = Fail
```

Jadi logika L1 bisa kita representasikan dengan list:

```
[L1_conj_tac, L1_mp_tac]
```

Akan tetapi, nantinya akan lebih memudahkan kalau kita merepresentasikan logika dengan tipe `L a` dimana `a` adalah bahasa yang menjadi basis dari logika tersebut. Misalnya untuk L1 kita definisikan:

```
data L1 a = L1 [Tactic a a]

L1_concrete :: L1 A1

L1_concrete
  =
  L1 [L1_conj_tac, L1_mp_tac]
```

Dimana `L1_concrete` adalah representasi nyata dari L1, sementara tipe L1 bisa dilihat sebagai enkapsulasi L1 sebagai tipe.

Menggunakan kelas, sekarang kita bisa memberikan *interface* yang seragam untuk mengakses logika. Misalnya kelas ini:

```
class Logic (L a) where

info :: L a -> String
```

menyediakan sebuah *interface*, yaitu fungsi `info` yang tujuannya untuk mengekstrak informasi informal dari sebuah logika. Sebagai contoh, implementasi dari *interface* ini untuk tipe `L1 A1` adalah:

```
instance Logic (L1 A1)

where

info _ = "Logika L1 adalah..."
```

Misalkan `f,g::Goal a` dan `t` adalah sebuah taktik dari sebuah logika *L* sedemikian sehingga:

```
f = t g
```

Maka `f|-g` adalah sebuah teorema dalam logika *L* dan `t` adalah buktinya. Taktik ini bisa merupakan taktik yang langsung diambil dari *L* (taktik primitif), atau merupakan komposisi dari taktik-taktik primitif. Contoh dari komposisi basis dari taktik adalah operasi THEN dan ORELSE. Komposisi ini dapat didefinisikan sebagai fungsi polimorfis yang bekerja uniform untuk semua logika:

```
THEN :: Tactic a b ->Tactic b c
        ->
        Tactic a c

(t THEN u) g = (u . t) g


ORELSE::Tactic a b ->Tactic a b
        ->
        Tactic a b

(t ORELSE u) g = case t g
     {OK f  -> OK f
      Fail  -> u g}
```

Efek dari `t THEN u` ke goal `g` adalah menerapkan taktik `t` dulu ke `g` kemudian baru `u`; sedangkan dalam `t ORELSE u` kita mencoba untuk menerap-kan `t` ke `g` dulu, apabila hasilnya `Fail`, maka kita mencoba `u`. Lihat misalnya HOL [4] yang memiliki koleksi operasi komposisi taktik yang cukup besar.

Sebuah teorema dapat direpresentasikan dengan pasangan `(f,g)` dimana `f,g::Goal a`. Beberapa implementasi logika, misalnya HOL, menyediakan tipe khusus dengan akses terbatas. Tujuannya adalah supaya user tidak bisa sembarangan menciptakan teorema. Sebuah teorema

hanya bisa diperoleh lewat aplikasi fungsi-fungsi tertentu yang dianggap 'aman', hal ini dibutuhkan untuk menjamin konsistensi dari logika tersebut.

6. MODULARITAS

Logika-logika sering kali memiliki operasi yang sama. Misalnya, kebanyakan logika mengijinkan unifikasi. Contoh lain adalah kedua hukum inferensi dari `L1` (`conj_tac` dan `mp_tac`) yang juga dikenal di banyak logika lain seperti logika proposisi dan logika predikat.

Dengan menggunakan kelas kita bisa menyediakan *interface* yang seragam untuk operasi-operasi yang secara konseptual di-*share* oleh beberapa logika. Sebagai contoh, berikut ini kita akan mendefinisikan signature `BasicLogic` dan signature `ConnectiveLogic`.

Coba lihat hukum inferensi berikut ini:

$$\frac{x > 0}{x\text{-}1 \geq 0} \quad (5)$$

Kita dapat menyimpulkan bahwa apabila a+1>0 maka a+1-1≥0. Yang sebetulnya kita lakukan adalah mencocokkan formula a+1>0 dengan kalimat x>0. Kita mencari substitusi dari variabel-variabel di x>0 sehingga dengan substitusi itu kita bisa memperoleh a+1>0 dari x>0. Dalam contoh ini substitusi yang diperlukan adalah mengganti x dengan a+1. Lalu kita gunakan substitusi ini terhadap x-1≥0 untuk mendapatkan hasil dari inferensi. Substitusi yang dimaksud diatas disebut juga *unifier*, dan proses mencari dan melakukan substitusi seperti diatas disebut *unifikasi*. Dengan tehnik yang sama kita juga bisa mereduksi goal x+1-1≥0, berdasarkan hukum inferensi diatas, menjadi x+1>0.

Berikut ini contoh dari dua kelas yang menyediakan *interface* seragam untuk logika.

```
class BasicLogic (L a)
where
unify      : a->a->a->Try a

class ConnectiveLogic (L a)
where
conj_tac :: Tactic a a
mp_tac   :: Tactic a a
```

Kelas `BasicLogic` menyediakan antara lain fungsi `unify`. Sesuai dengan namanya, fungsi `unify` melakukan unifikasi. Apabila `f`, `f'`, dan `g` adalah kalimat berbahasa `a`, maka:

```
unify f f' g'
```

akan mencari sebuah unifier supaya `f` bisa diubah menjadi `f'`; kemudian unifier ini diterapkan ke `g'` dan hasilnya merupakan keluaran dari fungsi tersebut.

Kelas `ConnectiveLogic` yang menyediakan beberapa *interface* untuk logika-logika yang memiliki konektif seperti ∧ dan ⇒.

Kita juga dapat melihat kelas sebagai kelompok logika-logika tertentu yang memiliki *interface* sama

Sebaliknya, sebuah logika bisa saja memiliki lebih dari satu *interface*.

Sebagai contoh, implementasi dari *interface-interface* diatas untuk logika `L1` strukturnya adalah seperti ini:

```
instance BasicLogic (L1 A1)
where
unify f g h = ...

instance ConnectiveLogic(L1 A1)
where
conj_tac = L1_conj_tac
mp_tac   = L1_mp_tac
```

Sebagai contoh, sekarang kita dapat mendefinisikan varian dari `mp_tac` yang juga melakukan unifikasi:

```
match_mp_tac ::
  (BasicLogic (L a),
   ConnectiveLogic (L a))
   =>
   Tactic a

match_mp_tac(Ok [A1_Imp p q,g])
  =
  {case unify q g f
   Ok f' -> mp_tac [f',g]
   Fail  -> Fail
  }

  where f = A1_Imp p q

match_mp_tac _ = Fail
```

Perhatikan bahwa fungsi ini bekerja pada semua logika yang mengimplementasi kelas `BasicLogic` dan `ConnectiveLogic`.

7. KOMPOSISI LOGIKA

Salah satu aspek penting dari sebuah logika adalah *soundness*. Sebuah logika disebut sound apabila semua taktiknya bisa dibuktikan konsisten relatif terhadap '*semantik*' tertentu. Sebuah logika tentunya tidak berguna, atau malah berbahaya untuk digunakan apabila ia tidak sound. Semantik adalah pengartian dari kalimat-kalimat dalam logika tersebut. Untuk membuktikan soundness dari sebuah logika kita biasanya membutuhkan pendeskripsian yang akurat dari semantik logika tersebut. Ini dilakukan dengan mendeskripsikan semantik tersebut dengan formula matematis. Semantik yang seperti ini disebut semantik formal.

Karena logika merupakan struktur formal, kita dapat menggunakan logika sebagai semantik dari logika lain. Ini sebetulnya cukup normal, karena sebuah logika memiliki bahasa tertentu untuk memformulasikan kalimat-kalimatnya. Bahasa ini dapat kita jadikan sebagai semantik dari bahasa dari logika lain.

Sebagai contoh, coba lihat bahasa A2 berikut ini.

```
type A2 = [A2_Clause]
type  A2_Clause = [A2_Atom]
data A2_Atom =
     A2_Const Bool
   | A2_Neg   String
   | A1_Pos   String
```

Bahasa ini dapat digunakan untuk merepresentasi-kan *Disjunctive Normal Form* (DNF) [9] dari sebuah kalimat proposisi. Sebagai contoh:

```
[[A1_Pos "p", A1_Ned "q"],
 [A1_Pos "r"],
 [A1_Const False]]
```

adalah representasi dari formula:

$$(p \wedge \neg q) \vee r \vee F$$

Kita bisa menggunakan kalimat-kalimat dari logika lain sebagai semantik formal. Formula dalam bentuk DNF lebih mudah untuk dibuktikan validitasnya, misalnya dengan menggunakan prosedur resolusi atau MESON [10].

Lihat logika `L2` berikut ini. Logika ini berbasis `A2` dan menyediakan dua taktik untuk membuktikan validitas sebuah DNF dengan menggunakan resolusi atau MESON.

```
data L2 a = L2 [Tactic a a]

L2_concrete :: L2 A2
L2_concrete
  =
 [res_tac, meson_tac]

 where
 res_tac   = ...
 meson_tac = ...
```

Setiap formula proposisi dapat diubah menjadi DNF yang equivalen. Lihat [10] untuk detil dari translasi ini. Jadi, kita bisa menulis fungsi `toDNF` berikut ini untuk menterjemahkan `L1` ke `L2`:

```
toDNF::L1->L2
toDNF = ...
```

Fungsi ini sering disebut juga sebagai fungsi semantik. Pada dasarnya fungsi ini adalah sebuah kompilator, karena ia menterjemahkan kalimat dari bahasa yang satu ke bahasa yang lain.

Ingat bahwa kita menggunakan notasi seperti ini untuk memformulasikan taktik dari sebuah logika:

$$\frac{g}{f} \qquad (6)$$

Notasi ini sering digunakan, walaupun memang tidak semua taktik mudah dinyatakan dalam bentuk diatas. Seringkali, taktik seperti ini dapat juga dinyatakan sebagai kalimat berbentuk $f \Rightarrow g$ dari logika tersebut. Ini memungkinkan kita untuk menganalisa taktik sebagaimana kita menganalisa kalimat-kalimat biasa dalam bahasa tersebut.

Sebagai contoh, kedua taktik dari L1 dapat ditulis dalam formula L1 sendiri, yaitu:

```
conj_tac: p∧q ⇒ p∧q
mp_tac  : (p⇒q)∧p ⇒ (p⇒q)∧q
```

Kebenaran atau validitas dari kedua taktik tersebut sekarang dapat dibuktikan dengan menterjemahkannya ke `L2` dengan menggunakan fungsi `toDNF`, lalu memanggil fungsi `reduce_tac` atau `meson_tac` untuk melakukan pembuktian di dalam `L2`.

Secara umum, andaikan `L1` dan `L2` adalah dua logika, yang satu berbasis bahasa `a` yang lain bahasa `b`. Andaikan `m` adalah fungsi semantik yang menginterpretasikan kalimat-kalimat dari `a` sebagai kalimat-kalimat dari `b`. Jadi:

```
m :: a->b
```

Apabila sebuah taktik t dari L1 dapat dinyatakan sebagai kalimat f_t dari a, maka konsistensinya dapat dibuktikan apabila f_t dapat direduksi menjadi list kosong lewat L2. Jadi, dengan kata lain apabila ada sebuah taktik v dari L2 sehingga:

```
v (Ok [m  f_t]) == []
```

Jadi fungsi verify untuk melakukan tugas verifikasi ini dapat ditulis sebagai berikut:

```
verify::Tactic b b->a->Bool
verify v f
=
(map' m THEN v) goal == Ok []
where
goal = Ok [f]
```

Fungsi map' adalah variasi dari map yang khusus bekerja pada goal:

```
map' f (Ok g) = Ok (map f g)
map' _ _      = Fail
```

Sebaliknya, semua taktik t dari L1 yang bisa ditulis dalam bentuk seperti dibawah ini pasti sound relatif terhadap semantik m, selama u dan v sendiri juga sound.

```
t = u THEN map' m THEN v
```

Sebagai contoh, kita dapat menulis taktik taut_tac dari logika L1 seperti ini:

```
taut_tac
=
map' toDNF
THEN
(res_tac ORELSE meson_tac)
```

Taktik ini melakukan transformasi ke logika L2 dengan representasi DNF-nya, lalu mencoba taktik res_tac atau meson_tac dari L2.

8. PENALARAN META

Bahasa kita gunakan untuk memformulasikan secara formal sifat tertentu dari obyek-obyek dunia nyata. Logika kita gunakan untuk membuktikan sifat tersebut. Logika sendiri juga merupakan obyek yang bisa kita analisa. Bagaimana apabila kita ingin membuktikan sifat tertentu dari sebuah logika L ? Dengan sendirinya kita membutuhkan logika lain dimana L bisa direpresentasikan dan dimana kita bisa melakukan penalaran yang dibutuhkan. Penalaran seperti ini disebut juga penalaran meta (*meta reasoning*) atau penalaran orde kedua. Merepresentasikan logika L di dalam logika lain sering juga disebut *embedding*.

Sebagai contoh dari persoalan sederhana yang membutuhkan penalaran meta, coba kita lihat dua fungsi berikut ini yang bekerja pada bahasa A1:

```
strip (A1_not p) = strip p
strip p          = p
even(A1_not(A1_not p)) = even p
even(A1_not p)         = False
even p                 = True
```

Jadi, fungsi strip membuang deretan operator ¬ terluar, sedangkan fungsi even mencek apakah deret tersebut mengandung ¬ yang jumlahnya genap. Kita inginmembuktikan bahwa apabila begitu halnya, maka f dan strip f equivalen. Jadi, kita ingin membuktikan dalam L1 bahwa:

```
∀f. even f
  ⇒
  (f ⇒ strip f ∧ stip f ⇒ f)
```

Persoalan ini sederhana, tetapi kita tidak bisa membuktikannya menggunakan L1 sendiri untuk membuktikan ini, walapun dengan modifikasi sederhana kita masih bisa membuktikan sifat diatas untuk kalimat-kalimat dari A1 secara individu. Akan tetapi, sifat diatas memerlukan quantifikasi universal atas semua kalimat dari L1 yang jumlahnya tak berhingga. Jelas kita tidak bisa memeriksa kalimat-kalimat A1 satu persatu, jadi alternatif yang tersisa adalah menganalisa definisi deklaratif atau 'source code' dari L1, even, dan strip karena source code tersebut besarnya berhingga.

Sejauh ini souce code yang kita gunakan untuk memodelkan logika serta semantiknya ditulis menggunakan sebuah subset dari bahasa Haskell. Untuk melakukan penalaran meta, seperti pembuktian sifat strip-even diatas, kita membutuhkan sebuah logika yang cukup ekspresif untuk bisa mengekspresikan kalimat-kalimat dari sub Haskell yang kita gunakan sebagai obyek dari logika tersebut. Salah satu contoh dari logika seperti ini adalah HOL.

Definisi dari L1, even, dan strip bisa kita masukan (tanpa banyak perubahan) ke HOL. Formula diatas, yang menspesifikasikan sebuah sifat dari strip dan even juga dapat diekspresikan dalam HOL, dan kemudian dibuktikan dengan melakukan induksi ke struktur dari bahasa A1.

Contoh-contoh lain dari persoalan-persoalan yang membutuhkan penalaran meta adalah:

1. Membuktikan ekuivalensi dari dua fungsi semantif.
2. Membuktikan sifat tertentu dari fungsi semantik, misalnya bahwa fungsi tersebut biyektif.
3. Membuktikan sifat tertentu dari bahasa yang merupakan basis dari sebuah logika, misalnya bahwa bahasa tersebut bersifat LL(k).
4. Membuktikan sifat tertentu dari taktik dari sebuah logika, misalnya terminasi dari taktik tersebut.

9. KESIMPULAN

Pendekatan yang kami gunakan berfokus pada konsep bahwa logika dapat dilihat sebagai sebuah bahasa dengan sekumpulan transformasi dalam bahasa tersebut ataupun ke bahasa lain. Berdasarkan konsep ini telah diperlihatkan bagaimana sebuah mekanisasi logika dilakukan dalam bahasa fungsional. Contoh yang diberikan memang sederhana, tetapi cukup untuk memberikan gambaran bahwa dengan konsep diatas sebuah logika dapat disusun secara modular dari sub-sub logika sehingga mekanisasi dari sebuah sistem multi logika menjadi lebih sederhana.

Aspek utama logika yaitu *soundness* telah dibahas, dan telah disusun sebuah rumusan untuk melakukan pengujian *soundness* dari taktik-taktik yang diimplementasikan. Aspek *soundness* tersebut dinyatakan relatif terhadap semantik dari logika dimana semantik tersebut juga dinyatakan dalam logika lain. Kondisi ini merupakan contoh dari *penalaran meta*.

Walaupun penalaran meta ini membutuhkan sebuah mekanisasi logika yang cukup ekspresif, namun demikian penyusunan logika dalam sebuah data tipe dapat dipetakan ke HOL dan selanjutnya proses penalarannya dapat dilakukan didalam HOL. Perbedaan utama dengan beberapa pendekatan penalaran meta yang ada, seperti yang diterapkan dalam theorem prover *Isabelle* adalah, dalam pendekatan ini kami tidak secara eksplisit menyatakan mana logika yang merupakan *object logic* dan mana yang *meta logic*. Pendekatan ini memungkinkan setiap logika memiliki kesempatan yang sama untuk dapat menjadi *meta logic*. Sebuah *object logic* tidak secara eksplisit dibangun dari *meta logic* melainkan dibangun dari komposisi dan instantiasi dari satu atau lebih logika lainnya. Pendekatan ini mengarah pada sebuah cara baru dalam pembuatan *generic theorem prover* dimana logika merupakan *library-library* yang dapat di-*plug-in* dengan mudah dan independent serta dapat digunakan dengan lebih fleksible. Lebih jauh lagi, dapat memudahkan dalam memverifikasi sistem non-trivial yang membutuhkan penerapan multi logika.

Suatu aturan main yang lebih konkret dibutuhkan untuk dapat menjadikan konsep ini dapat diterapkan dalam pengembangan *theorem prover*. Aturan-aturan main tersebut seperti aturan standarisasi dalam pengembangan *library*. Dibutuhkan suatu konsep klasifikasi kelas-kelas logika, dan standarisasi dari fungsi-fungsi *interface* dari logika tersebut. Untuk dapat melakukan penalaran meta, sebuah standarisasi untuk pembuatan fungsi-fungsi logika juga dibutuhkan, sehingga pemetaan dapat dilakukan secara automatis.

REFERENSI

[1] Lawrence C. Paulson, *ML for the Working Programmer*, 2nd Edition, Cambridge University Press, 1996.

[2] Markus Kaltenbach, Interactive Verification Exploiting Program Design Knowledge: A Model-Checker for UNITY, *Technical Report no. CS-TR-96-22*, The University of Texas, Austin, USA, 1997.

[3] K.M. Chandy and J. Misra, *Parallel Program Design*, Austin, Texas, Addison-Wesley, May 1989.

[4] M.J.C. Gordon and T.F. Melham, *Introduction to HO*,. Cambridge University Press, 1993.

[5] Flemming Anderson, *A Theorem Prover for UNITY in Higher Order Logic*. PhD Thesis, Technical University of Denmark, 1992.

[6] I.S.W.B. Prasetya, *Mechanically Supported Design of Self-stabilizing Algorithms*. PhD thesis, Utrecht University, 1995.

[7] Lawrence C. Paulson, Isabelle: the next 700 theorem provers. in: P. Odifreddi (editor), *Logic and Computer Science* ,Academic Press, 1990.

[8] A.J.T. Davie: *An Introduction toFfunctional Programming Systems Using Haskell*. Cambridge university press, 1992.

[9] Kenneth H. Rosen(editor), *Handbook of Discrete and Combinatorial Mathematics*, Boca Raton [etc.]: CRC Press, 2000.

[10] John Harrison, Optimizing Proof Search in Model Elimination. In M. A. McRobbie and J. K. Slaney, editors, 13th *International Conference on Automated Deduction, volume 1104 of Lecture Notes in Computer Science*, pages 313--327, New Brunswick, NJ, Springer-Verlag, 1996.

[11] Per Bjesse, *Automatic Verification of Combinational and Pipelined FFT Circuits*, CAV'99, Springer-Verlag, 1999.