Bab 2

Definite Clause Grammar (DCG)

Bab ini memaparkan tentang DCG, fitur PROLOG yang digunakan sebagai *parser* dalam implementasi *principal-type algorithm* dan algoritma pencarian *type inhabitant*. Bab ini dimulai dengan penjelasan *Context Free Grammar* (CFG). Setelah itu, dijelaskan pengertian dari DCG. Kemudian bab ini dilanjutkan dengan penjelasan kemampuan (fitur) dari DCG, beserta cara penggunaannya.

2.1 Context Free Grammar (CFG)

CFG adalah suatu 4-tupple (*quadruple*) G = (V, Σ, S, P), dimana [Mar03]:

- V merupakan himpunan *variable* atau simbol *non-terminal*.
- Σ merupakan himpunan simbol *terminal*.
- V dan Σ (himpunan simbol) ini merupakan himpunan berhingga yang *disjoint*.
- S adalah suatu unsur dari V, yang disebut juga dengan *start-symbol*.
- P adalah himpunan berhingga dari formula berbentuk $A \rightarrow \alpha$, dimana $A \in V$ dan $\alpha \in (V \cup \Sigma)^*$. Unsur dari P ini disebut sebagai *grammar rules* atau *productions*.

Contoh:

Spesifikasi CFG untuk membangkitkan bahasa yang terdiri dari semua *palindrom* atas {a,b,c} adalah $G_1 = (V_1,\Sigma_1,S_1,P_1)$:

- $V_1 = \{S_1\}$,

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

- Σ_1= {a,b,c},
- S_1 = *start-symbol*,
- P_1 = { $S_1 \rightarrow \lambda$,

  $S_1 \rightarrow a$,

  $S_1 \rightarrow b$,

  $S_1 \rightarrow c$,

  $S_1 \rightarrow aS_1a$,

  $S_1 \rightarrow bS_1b$,

  $S_1 \rightarrow cS_1c$ }.

($S_1 \rightarrow a$, artinya adalah setiap kali S_1 muncul dapat ditulis ulang dengan a).

Dari CFG tersebut, kita dapat melihat bahwa beberapa contoh urutan simbol yang dibangkitan adalah abba, acca, ababa, abcba, abbabba, yang semuanya merupakan *palindrom* atas {a,b,c}. Untuk urutan simbol abba, dibangkitkan dengan dimulai dari "*start-symbol*", lalu diterapkan *production rule* ke 5, yaitu $S_1 \rightarrow$ "aS_1a". Dari penerapan *production rule* tersebut, diperoleh urutan simbol aS_1a. Kemudian langkah selanjutnya adalah menerapkan *production rule* no 6, yaitu $S_1 \rightarrow bS_1b$, dan dari penerapan *production rule* tersebut, diperoleh urutan simbol "abS_1ba". Lalu, langkah yang terakhir adalah menerapkan *production rule* no 1, yaitu $S_1 \rightarrow \lambda$. Akhirnya, diperolehlah urutan simbol "abba". Jika ditulis secara singkat langkah-langkah tersebut menjadi:

$$S_1 \Rightarrow aS_1a \Rightarrow abS_1ba \Rightarrow abba.$$

Langkah-langkah ini merupakan langkah untuk membangkitkan urutan simbol yang merupakan bagian dari suatu bahasa, yaitu dengan bermula pada *start-symbol*, dan kemudian terapkan *production rule* yang telah dispesifikasikan.

Proses pembangkitan suatu urutan simbol tersebut disebut dengan **derivasi**. Kemudian himpunan dari urutan simbol yang dibangkitkan *grammar* tersebut akan membentuk **bahasa**.

Dari penjelasan tersebut, kita dapat melihat bahwa *grammar* bisa digunakan untuk membangkitan kalimat (***generate sentence***), misalnya aabbaa, aba, dll. Kemudian, himpunan kalimat tersebut disebut bahasa yang didefinisikan oleh grammar yang

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

bersangkutan. Salah satu kegunaan *grammar* yang penting adalah untuk mengenali kalimat (***recognize sentence***). Sebagai pengenal kalimat, *grammar* berfungsi untuk menentukan apakah sebuah kalimat dapat dibangkitkan oleh *grammar* tersebut. Secara garis besar, proses pengenalan ini adalah kebalikan dari proses pembangkitan. Adapun prosesnya adalah sebagai berikut [Bra01]:

1. Proses pengenalan diawali dari urutan simbol yang diberikan.

2. Terapkan *grammar rules* pada urutan simbol tersebut. Namun penerapan tersebut dilakukan secara terbalik. Jika urutan simbol tersebut mengandung sub-bagian urutan simbol yang cocok dengan sisi kanan dari suatu *rule* pada *grammar rules*, maka ganti sub-bagian dari urutan simbol tersebut dengan sisi kiri dari *rule* tersebut.

3. Proses pengenalan berakhir ketika urutan simbol yang diberikan telah selesai menjadi *start-symbol*.

4. Jika tidak ada cara untuk menghasilkan *start-symbol* dengan cara di atas, maka urutan simbol tersebut bukanlah termasuk urutan simbol yang bisa dibangkitkan oleh *grammar* tersebut.

Inti dalam proses pengenalan ini adalah kalimat dipecah menjadi unsur pembentuknya, proses ini disebut *parsing*.

Untuk mengenali apakah suatu kalimat dapat dibangkitkan dari suatu *grammar*, perlu dibuat suatu program *parsing* untuk *grammar* yang bersangkutan. Dalam PROLOG hal ini menjadi mudah dengan adanya fitur DCG, karena *grammar* yang ditulis dalam DCG sudah merupakan *parser* untuk *grammar* tersebut [Bra01].

2.2 Pengertian DCG

DCG merupakan fitur dalam PROLOG yang digunakan untuk mengekspresikan *grammar rules* (aturan *grammar*). DCG ini juga bisa disebut sebagai generalisasi dari CFG yang *executable*, yang ditambahkan fitur dari PROLOG. Dalam mengekpresikan *grammar rules*, DCG menggabungkan kemampuan dari PROLOG (seperti *Unification*, dan

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

pemanggilan terhadap predikat *built-in*) dan CFG. Dengan penggabungan tersebut, meningkatkan kemampuan DCG dalam mengekspresikan suatu *grammar* [ES94].

2.3 Kemampuan dan Fitur DCG

Seperti yang telah dijelaskan sebelumnya, DCG memiliki kemampuan untuk mengekspresikan CFG. Selain itu, terdapat pula beberapa kemampuan lain dari DCG seperti fitur untuk memberikan argumen pada simbol *non-terminal* dan juga kemampuan untuk memanggil predikat dalam PROLOG. Dengan adanya fitur-fitur ini, kemampuan DCG untuk mengekspresikan suatu *grammar* menjadi semakin meningkat. Berikut adalah pembahasan satu persatu dari fitur DCG tersebut.

2.3.1 Merepresentasikan CFG

DCG dapat digunakan untuk menspesifikasikan CFG dalam suatu program PROLOG. Untuk menspesifikasikan suatu CFG menjadi DCG, kita cukup mengubah beberapa notasi penulisan. Adapun perubahannya adalah sebagai berikut[Bra01]:

1. Notasi "→" dalam CFG, dituliskan menjadi "`-->`" dalam DCG di PROLOG.
2. Notasi simbol *terminal* dituliskan di dalam kurung kotak "[ ]".
3. Notasi simbol *non-terminal* dituliskan langsung dengan huruf kecil.
4. Dalam DCG, antar simbol dibatasi dengan koma ','.
5. Setiap *rule* dalam DCG diakhiri dengan tanda titik '.'.

Sebagai contoh pertama, misalnya kita memiliki spesifikasi CFG yang membangkitkan bahasa a*r*i*o*. Dengan kata lain adalah bahasa yang terdiri dari himpunan urutan simbol, yang terdiri dari sejumlah simbol a kemudian diikuti sejumlah simbol r, kemudian diikuti sejumlah simbol i, dan diakhiri dengan sejumlah simbol o. Contohnya adalah aarriioo, arrii, ario, dll. Adapun spesifikasi CFG yang membangkitkan bahasa tersebut adalah:

- V = {S,A,R,I,O},

- $\Sigma = \{a,r,i,o\}$,
- S = *start-symbol*,
- P = { $S \rightarrow A\ R\ I\ O$,
  $A \rightarrow a\ A$,
  $R \rightarrow r\ R$,
  $I \rightarrow i\ I$,
  $O \rightarrow o\ O$,
  $A \rightarrow \lambda$,
  $R \rightarrow \lambda$,
  $I \rightarrow \lambda$,
  $O \rightarrow \lambda$, }

Kemudian dari CFG tersebut kita ubah menjadi DCG, hasilnya adalah sebagai berikut:

```
s --> a, r, i, o.
a --> [a],a.
a --> [].
r --> [r],r.
r --> [].
i --> [i],i.
i --> [].
o --> [o],o.
o --> [].
```

Dalam PROLOG, setelah menuliskan DCG tersebut, kita dapat langsung menggunakan program tersebut sebagai *parser* atau program untuk mengenali suatu urutan simbol. Untuk menggunakan *parser* atau program pengenalan tersebut, sekumpulan simbol yang diberikan harus direpresentasikan dalam bentuk *difference list* dari sekumpulan simbol *terminal* [Bra01]. Misalnya, ***ario*** dapat direpresentasikan sebagai:

- List $[\boldsymbol{a}, \boldsymbol{r}, \boldsymbol{i}, \boldsymbol{o}]$ dan list $[]$.
- List $[\boldsymbol{a}, \boldsymbol{r}, \boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}]$ dan list $[\boldsymbol{s}]$.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

- List $[\boldsymbol{a, r, i, o, 1, 0, 1}]$ dan list $[\boldsymbol{1, 0, 1}]$.

- List $[\boldsymbol{a, r, i, o, s, a, n, t, o, s, o}]$ dan list $[\boldsymbol{s, a, n, t, o, s, o}]$.

Selanjutnya kita bisa memberikan *query* pada program PROLOG dengan menggunakan *start-simbol* sebagai nama predikat. Kemudian sekumpulan simbol yang diberikan sebagai argumen harus direpresentasikan dalam bentuk *difference list* dari sekumpulan simbol *terminal*. *Query* seperti ini dapat digunakan untuk mengetahui apakah suatu urutan simbol tersebut dapat dibangkitkan oleh *grammar* yang telah dispesifikasikan sebagai DCG dalam program PROLOG tersebut. Berikut adalah contoh penggunaan program DCG untuk mengenali pola urutan simbol yang memiliki pola a*r*i*o*:

```
?- s([a,a,r],[]).
yes
?- s([i,a,r],[]).
no
?- s([a,a,r,i],[]).
yes
?- s([a,r,i,o,d,n,s],[d,n,s]).
yes
?- s([a,a,r,i,a,r,i,o],[a,r,i,o]).
yes
```

Selain itu, kita bisa juga menggunakan program tersebut untuk membangkitkan suatu urutan simbol, dengan cara memberikan variable pada *query*. Contohnya adalah sebagai berikut:

```
?- s([A,B,C],[]).
A = a,
B = a,
C = a ;

A = a,
B = a,
C = r
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

⋮

Contoh yang kedua adalah, misalkan kita ingin membuat *grammar* yang menspesifikasikan susunan urutan gerakan yang *valid* untuk suatu robot (Contoh ini diadaptasi dari [Bra01]). Robot tersebut hanya dapat bergerak ke atas dan ke bawah. Adapun DCG-nya adalah sebagai berikut:

```
move --> step.
move --> step, move.
step --> [up].
step --> [down].
```

Dalam DCG di atas, dispesifikasikan bahwa gerakan robot (*move*) bisa merupakan sebuah *step*, yang terdiri dari "up" dan "down". Lalu gerakan robot (*move*) yang lain juga bisa merupakan kombinasi sebuah *step* dan *move* itu sendiri. Hal tersebut mendefinisikan gerakan robot secara rekursif. Berikut ini adalah contoh *query* dalam menggunakan program PROLOG tersebut untuk mengenali apakah suatu rangkaian (urutan) gerakan robot *valid* atau tidak:

```
?- move([up, up, down],[]).
yes
?- move([up, left, down],[]).
no
?- move([up, STEP, down],[]).
STEP = up ;
STEP = down ;
No
```

Kemudian contoh yang ketiga adalah mengenai penggunaan *grammar* dalam bahasa *natural* (Contoh ini diadaptasi dari [Bra01] dengan mengalami sedikit perubahan). Misalnya kita ingin membuat program untuk mengenali apakah suatu kalimat sesuai dengan struktur yang benar atau tidak, dan struktur kalimat yang benar adalah sebagai berikut:

- Sebuah kalimat tersusun dari *noun phrase* dan *verb phrase*.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

- Sebuah *verb phrase* terdiri dari *verb* dan *noun phrase*.
- Sebuah *noun phrase* terdiri dari *determiner* dan *noun*.
- Misalkan untuk kasus ini:
  - kita memiliki 2 *determiner* yaitu "a" dan "the"
  - kita memiliki 2 *noun* yaitu "cat" dan "mouse"
  - kita memiliki 2 *verb* yaitu "scares" dan "hates"

Dari spesifikasi aturan di atas, jika kita tuliskan dalam bentuk DCG, maka kita peroleh spesifikasi sebagai berikut:

```
sentence --> noun_phrase, verb_phrase.
verb_phrase --> verb, noun_phrase.
noun_phrase --> determiner, noun.
determiner --> [a].
determiner --> [the].
noun --> [cat].
noun --> [mouse].
verb --> [scares].
verb --> [hates].
```

Beberapa contoh kalimat yang dikenali dan dibangkitkan oleh spesifikasi DCG tersebut adalah:

- `[the, cat, scares, a, mouse]`
- `[the, mouse, hates, the, cat]`
- `[the, mouse, scares, the, mouse]`

Hingga saat ini, kalimat yang dibangkitkan masih merupakan kalimat *singular*, karena *verb* dan *noun* yang ada hanyalah untuk kalimat *singular*. Untuk itu, sekarang ditambahkan *verb* dan *noun* untuk *plural*, sehingga DCG tersebut bisa membangkitkan kalimat *plural* seperti [the, mice, hate, the, cats]. Spesifikasi *grammar*-nya adalah sebagai berikut:

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
sentence --> noun_phrase, verb_phrase.
verb_phrase --> verb, noun_phrase.
noun_phrase --> determiner, noun.
determiner --> [a].
determiner --> [the].
noun --> [cat].
noun --> [mouse].
noun --> [cats].
noun --> [mice].
verb --> [scares].
verb --> [hates].
verb --> [scare].
verb --> [hate].
```

Dengan penambahan *verb* dan *noun* seperti di atas, kalimat seperti [the, mice, hate, the, cats] dapat dikenali oleh DCG ini. Dengan kata lain, spesifikasi DCG yang sekarang bisa mengenali dan membangkitkan kalimat *plural.* Akan tetapi, spesifikasi DCG tersebut juga bisa mengenali dan membangkitkan kalimat yang tidak *valid* dalam aturan *grammar* bahasa inggris, misalnya [the, mouse, hate, the, cat]. Karena kalimat yang benar harusnya adalah [the, mouse, hates, the, cat]. Permasalahannya terletak pada *rule*:

```
"sentence --> noun_phrase, verb_phrase."
```

Dalam *rule* tersebut, dinyatakan bahwa ***noun_phrase*** dan ***verb_phrase*** dapat digabungkan langsung untuk membentuk kalimat tanpa perlu memperhatikan *context*-nya (*single* atau *plural*). Padahal dalam bahasa Inggris, ***noun_phrase*** dan ***verb_phrase*** harus memiliki *context* yang sama dalam penggunaannya (sama-sama *plural* atau sama-sama *singular*). Hal ini disebut *context-dependent* [Bra01], yaitu setiap frase (*phrase*) yang muncul bergantung pada *context* kalimatnya, *singular* atau *plural.* Permasalahan ini tidak bisa ditangani dengan CFG, namun dengan DCG hal ini bisa ditangani dengan mudah, yaitu dengan menggunakan kemampuan DCG untuk menambahkan argumen pada simbol *non-terminal* pada spesifikasi *grammar.* Hal ini dijelaskan pada sub-bab 2.3.2.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

2.3.2 Pemberian Argumen Pada Simbol *Non-Terminal*

Dalam DCG, kita bisa memberikan argumen pada simbol *non-terminal.* Dengan kemampuan ini permasalahan yang dihadapi pada contoh ke tiga di sub-bab 2.3.1 dapat terselesaikan. Dalam menyelesaikan permasalahan tersebut, kita cukup memberikan argumen tambahan untuk memastikan bahwa ***noun_phrase*** dan ***verb_phrase*** memiliki *context* yang sama dalam penggunaannya (sama-sama *plural* atau sama-sama *singular*). Berikut adalah spesifikasi DCG yang telah diubah dengan ditambahkan argumen pada setiap simbol *non-terminal*:

```
sentence(Number) --> noun_phrase(Number), verb_phrase(Number).
verb_phrase(Number) --> verb(Number), noun_phrase(Number).
noun_phrase(Number) --> determiner(Number), noun(Number).
determiner(singular) --> [a].
determiner(singular) --> [the].
determiner(plural) --> [the].
noun(singular) --> [cat].
noun(singular) --> [mouse].
noun(plural) --> [cats].
noun(plural) --> [mice].
verb(singular) --> [scares].
verb(singular) --> [hates].
verb(plural) --> [scare].
verb(plural) --> [hate].
```

Penambahan argumen pada simbol *non-terminal* ini untuk memastikan agar ***noun_phrase*** dan ***verb_phrase*** selalu memiliki *context* yang sama dalam penggunaannya (sama-sama *plural* atau sama-sama *singular*). Dengan adanya penambahan satu argumen pada simbol *non-terminal* ini, *Query* yang digunakan untuk membangkitkan atau mengenali suatu kalimat jadi berbeda. Perbedaannya ini terletak pada penambahan isi argumen sebelum memberikan kalimat yang ingin di kenali. Untuk lebih jelasnya, kita bandingkan bentuk *query* sebelum ada penambahan argumen dan setelah ada penambahan argumen. Berikut adalah contoh *query* sebelum ada penambahan argumen pada simbol *non-terminal*:

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
?- sentence([the, mouse, hates, the, cat],[]).
yes
```

Berikut adalah contoh *query* setelah ada penambahan argumen pada simbol *non-terminal*:

```
?- sentence(singular,[the, mouse, hates, the, cat],[]).
yes
```

Dari contoh tersebut kita melihat bahwa kita perlu menambahkan argumen jenis kalimat (*singular* atau *plural*). Hal ini merupakan argumen pada simbol *non-terminal* dalam spesifikasi DCG tersebut. Dari sini kita bisa melihat bahwa pola *query* yang diberikan adalah:

```
<nama simbol non-terminal>(arg1,arg2,...,argN,[<kalimat>|<T>],[<T>]).
```

Contoh:

```
?- sentence(plural,[the, mouse, hates, the, cat],[]).
yes
```

Karena *start-simbol* kita adalah *sentence*, $\langle$ nama simbol non-terminal $\rangle$ = *sentence*. Kemudian karena *sentence* hanya memiliki satu argumen (yaitu jenis kalimat), pada *query* hanya diberikan satu argumen (dalam contoh ini adalah *plural*).

Selain menentukan apakah kalimat yang diberikan pada *query* sesuai dengan *grammar* atau tidak, program ini juga bisa menentukan jenis kalimat yang diberikan (apakah *plural* atau *singular*). Hal tersebut bisa dilakukan dengan memberikan *variable* pada bagian argumen di *query*. Berikut adalah contoh *query* dari penggunaan program ini sebagai pengenal kalimat, penentu jenis kalimat dan juga sebagai pembangkit kalimat yang sesuai dengan spesifikasi *grammar* yang diberikan:

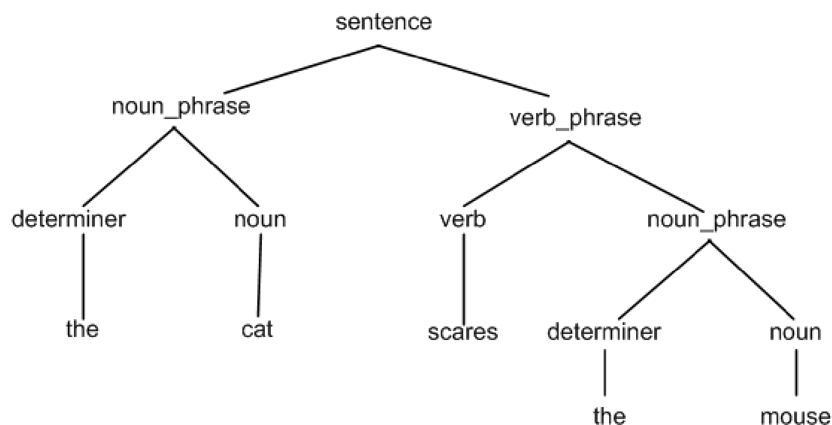
```
?- sentence(singular,[the, mouse, hates, the, cat],[]).
yes
?- sentence(plural,[the, mouse, hates, the, cat],[]).
no
?- sentence(Number,[the, mouse, hates, the, cat],[]).
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
Number = singular
?- sentence(plural,[the, mice, hate, the, cats],[]).
yes
?- sentence(singular,[the, mice, hate, the, cats],[]).
no
?- sentence(singular,[the, mouse, hate, the, cat],[]).
no
?- sentence(singular,[the, What, hates, the, cat],[]).
What = cat ;
What = mouse ;
no
```

Dari contoh penggunaan tersebut, dapat terlihat bahwa permasalahan yang terjadi pada contoh 3 di sub-bab 2.3.1 tidak terjadi lagi. Sekarang kalimat [the, mouse, hate, the, cat] dianggap salah, karena tidak ada kesesuaian *context* (*singular* atau *plural*). Dari sini kita bisa melihat bahwa fitur ini bisa meningkatkan kemampuan DCG untuk mengekspresikan *grammar rules*.

Berikut ini adalah contoh yang kedua dari penambahan argumen pada simbol *non-terminal*. Dalam contoh ini diperlihatkan bagaimana cara membuat *parse tree* (seperti yang diperlihatkan pada Gambar 2.1) yang merepresentasikan susunan kata dari kalimat yang dibangkitkan oleh *grammar* pada contoh pertama (Contoh ini diambil dari [Bra01]).



Gambar 2.1: *Parse tree* dari kalimat "the cat scares the mouse"

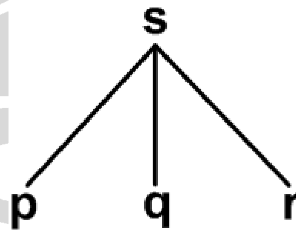
Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

Sebelum masuk pada spesifikasi DCG untuk membuat konstruksi *parse tree* tersebut, terlebih dahulu kita definisikan struktur *parse tree* tersebut. *Parse tree* dari sebuah frase (*phrase*) atau kalimat adalah sebuah *tree* dengan properti sebagai berikut:

- Semua *leaves* dari *tree* tersebut diberi nama dengan nama simbol *terminal* pada *grammar*.
- Semua *internal node* dari *tree* diberi nama dengan nama simbol *non-terminal*, dan *root* dari *tree* diberi nama dengan nama simbol *non-terminal* yang bersesuaian dengan nama jenis frasenya.
- relasi antar anak dan *parent* dalam *tree* dispesifikasikan (didefinisikan) sesuai dengan aturan pada *grammar*. Misalnya jika *grammar* mengandung aturan

```
s --> p,q,r
```

  maka *tree* akan mengandung *node* dengan nama "s" dan memiliki 3 anak bernama "p", "q",dan "r" (lihat Gambar 2.2).



Gambar 2.2: *Parse tree* dari rule *grammar* "s −− > p, q, r" [Bra01]

Berikut ini adalah spesifikasi DCG yang digunakan untuk membuat konstruksi *parse tree* yang merepresentasikan susunan kata dari kalimat yang dibangkitkan oleh *grammar* pada contoh pertama:

```
sentence(Number, sentence(NP,VP))
   --> noun_phrase(Number,NP), verb_phrase(Number,VP).
verb_phrase(Number, verb_phrase(Verb,NP))
   --> verb(Number,Verb), noun_phrase(Number,NP).
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
noun_phrase(Number, noun_phrase(Det,Noun))
   --> determiner(Number,Det), noun(Number,Noun).
determiner(singular, determiner(a)) --> [a].
determiner(singular, determiner(the)) --> [the].
determiner(plural, determiner(the)) --> [the].
noun(singular, noun(cat)) --> [cat].
noun(singular,  noun(mouse)) --> [mouse].
noun(plural, noun(cats)) --> [cats].
noun(plural, noun(mice)) --> [mice].
verb(singular, verb(scares)) --> [scares].
verb(singular, verb(hates)) --> [hates].
verb(plural, verb(scare)) --> [scare].
verb(plural, verb(hate)) --> [hate].
```

Berikut adalah contoh *query* dalam menggunakan program tersebut untuk membuat konstruksi *parse tree* yang merepresentasikan susunan kata dari kalimat yang diberikan:

```
?- sentence(singular,Tree,[the,cat,scares,the,mouse],[]).
Tree = sentence(noun_phrase(determiner(the), noun(cat)), verb_phrase
(verb(scares), noun_phrase(determiner(the), noun(mouse)))) ;
No
```

atau jika dilihat dalam bentuk gambar, maka akan tampak seperti Gambar 2.1.

Dari penjelasan pada sub-bab 2.3.2 dan 2.3.1, kita telah melihat bagaimana DCG dapat mengekspresikan CFG dalam program PROLOG. Kemudian spesifikasi DCG itu sendiri bisa langsung berfungsi sebagai program untuk mengenali atau membangkitkan urutan simbol yang *valid* sesuai dengan definisi yang diberikan dalam *grammar*. Akan tetapi, ada permasalahan spesifikasi *grammar* yang tidak dapat ditangani hanya dengan spesifikasi CFG biasa. Namun masalah ini bisa diatasi dengan memanfaatkan kemampuan tambahan yang dimiliki oleh DCG, yaitu kemampuan untuk memberikan argumen pada simbol *non-terminal*.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

2.3.3 Pemanggilan Predikat PROLOG

Fitur lain yang tak kalah pentingnya dari DCG adalah kemampuan memanggil predikat dalam PROLOG atau memasukkan sebuah *PROLOG goals* ke dalam bagian aturan *grammar*. Goals atau predikat PROLOG yang ingin dipanggil tersebut dituliskan di dalam kurung kurawal "{ }". Semua yang ada di dalam kurung kurawal ini dieksekusi seperti predikat PROLOG atau PROLOG *goal* seperti biasanya [Bra01] [ES94].

Untuk contoh pertama penggunaan fitur ini, kita lakukan modifikasi pada spesifikasi *grammar* yang mendefinisikan gerakan robot yang *valid* (lihat contoh kedua pada subbab 2.3.1). Berikut adalah spesifikasi awalnya:

```
move --> step.
move --> step, move.
step --> [up].
step --> [down].
```

Sekarang, selain ingin memiliki program untuk mengenali dan membangkitkan urutan gerakan yang *valid*, kita juga ingin menghitung jarak pergerakan robot. Kita definisikan bahwa gerakan ke atas akan menambahkan nilai jarak perpindahan dengan angka 1 dan gerakan ke bawah akan mengurangi nilai jarak perpindahan dengan angka 1. Perubahan yang dilakukan di sini adalah sebagai berikut:

- Menambahkan sebuah *variable* "*distance*" pada setiap simbol *non-terminal*.
- Untuk *step up*, "*distance*" = 1
  - step(1) −− > [up].
- Untuk *step down*, "*distance*" = -1
  - step(-1) −− > [down].
- Pada definisi rekursif dari simbol *non-terminal move* dilakukan penjumlahan aritmatika, di sinilah proses pemanggilan terhadap PROLOG *goals* berperan.
  - move(Distance) −− > step(D1), move(D2), {Distance is D1+D2}.

Berikut ini adalah spesifikasi lengkap DCG tersebut:

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
move(Distance) --> step(Distance).
move(Distance) --> step(D1), move(D2), {Distance is D1+D2}.
step(1) --> [up].
step(-1) --> [down].
```

Kemudian berikut ini adalah contoh *query* penggunaannya:

```
?- move(D,[up,up,up],[]).
D = 3
?- move(D,[up,up,up,down],[]).
D = 2
```

Untuk contoh yang kedua, kita modifikasi contoh pertama dari sub-bab 2.3.1, yaitu *grammar* untuk membangkitkan bahasa a*r*i*o*. Sekarang kita ingin menghitung jumlah semua huruf dalam suatu urutan simbol yang dibangkitkan *grammar* tersebut. Berikut ini adalah spesifikasi DCG-nya:

```
s(N) --> a(NA), r(NR), i(NI),  o(NO), {N is NA + NR + NI + NO}.
a(N) --> [a], a(NA),{N is NA+1}.
a(0) --> [].
r(N) --> [r], r(NR),{N is NR+1}.
r(0) --> [].
i(N) --> [i], i(NI),{N is NI+1}.
i(0) --> [].
o(N) --> [o], o(NO),{N is NO+1}.
o(0) --> [].
```

Berikut adalah contoh *query* penggunaannya:

```
?- s(N,[a,a,r,i,i,o],[]).
N = 6
?- s(N,[a,a,r,r,i,o,o],[]).
N = 7
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

Bab 3

Type-Free $\boldsymbol{\lambda}$-*Calculus*

Bab ini menjelaskan *type-free* $\boldsymbol{\lambda}$-*calculus*. Maksud *type-free* di sini adalah pembahasan pada bab ini hanya berfokus pada $\boldsymbol{\lambda}$-*calculus* tanpa menyinggung (membahas) tentang tipe pada $\boldsymbol{\lambda}$-*Calculus*. Penjelasan pada bab ini dimulai dengan pemaparan secara singkat mengenai $\boldsymbol{\lambda}$-*calculus*. Pembahasan bab ini hanya difokuskan pada bagian yang sederhana dari $\boldsymbol{\lambda}$-*calculus*, teori $\boldsymbol{\lambda}$-*calculus* yang murni, dengan $\boldsymbol{\lambda}$-*term* yang hanya terbentuk dari aplikasi, abstraksi, dan *term-variable*. Tidak hanya berhenti di situ, bab ini juga menjelaskan definisi-definisi yang berkaitan dengan $\boldsymbol{\lambda}$-*term*. Terakhir bab ini ditutup dengan penjelasan mengenai beberapa operasi pada $\boldsymbol{\lambda}$-*term*. Acuan utama dalam bab ini adalah [Hin97], termasuk definisi-definisi, teorema dan lemma yang dijelaskan pada bab ini. Selain itu, digunakan juga [Bar92] sebagai referensi penunjang.

3.1 Penjelasan Singkat $\boldsymbol{\lambda}$-*Calculus*

$\boldsymbol{\lambda}$-*calculus* merupakan sekeluarga bahasa pemrograman *prototype* yang dibuat oleh seorang pakar logika bernama Alonzo Church pada tahun 1930. Ciri utama dari $\boldsymbol{\lambda}$-*calculus* adalah sebagai berikut:

- $\boldsymbol{\lambda}$-*calculus* bersifat *Functional*, dengan kata lain $\boldsymbol{\lambda}$-*calculus* didasarkan pada ide (gagasan atau pemikiran) tentang fungsi atau operator. Selain itu, $\boldsymbol{\lambda}$-*calculus* juga memiliki notasi untuk aplikasi dan abstraksi dari fungsi (*function application* dan *abstraction*).

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

- $\boldsymbol{\lambda}$-*calculus* bersifat *Higher-order*, dengan kata lain $\boldsymbol{\lambda}$-*calculus* memberikan notasi sistematis untuk menggambarkan suatu fungsi (operator) yang input atau outputnya dapat berupa sebuah fungsi (operator) lain.

$\boldsymbol{\lambda}$-*calculus* memberikan mekanisme untuk mengekspresikan fungsi sebagai suatu aturan korespondensi antara argumen dan hasil fungsi secara formal. Fungsi disini maksudnya adalah seperti fungsi umum di matematika (contohnya $\boldsymbol{f(x) = m}$), atau misalnya di dalam bahasa pemrograman (contohnya *method* dalam bahasa pemrograman Java$^{\boldsymbol{TM}}$ dan *function* dalam Haskel). Berikut adalah contoh notasi penulisan abstraksi suatu fungsi dalam $\boldsymbol{\lambda}$-*calculus*:

$$\boldsymbol{\lambda x.M}$$

Bentuk tersebut merupakan bentuk umum suatu fungsi, dengan x adalah sebuah *formal parameter*, dan M adalah hasil (*ouput*) dari fungsi.

Lalu, bagaimana cara kita menghubungkan antara *formal parameter* dan *actual parameter*? dalam $\boldsymbol{\lambda}$-*calculus*, caranya adalah dengan membuat suatu aplikasi seperti berikut:

$$\boldsymbol{(\lambda x.M)N}$$

dengan x adalah sebuah *formal parameter*, M adalah hasil (*ouput*) dari fungsi, dan N adalah *actual parameter*. Kemudian untuk mengevaluasi aplikasi ini atau menerapkan *actual parameter* yang diberikan pada fungsi tersebut, kita lakukan substitusi untuk menggantikan *formal parameter* pada fungsi, dengan *actual parameter* yang diberikan seperti berikut ini:

$$\boldsymbol{[N/x]M}$$

dengan x adalah sebuah *formal parameter*, M adalah hasil (*ouput*) dari fungsi, dan N adalah *actual parameter*. Sebagai contoh konkrit, misalnya kita memiliki fungsi sebagai berikut:

$$\boldsymbol{(\lambda x.x^2 - 10)}$$

Kemudian kita memiliki *actual parameter* N = 6. Untuk menerapkan *actual parameter* pada fungsi tersebut, kita buat aplikasi dan lakukan substitusi untuk menggantikan *formal parameter* pada fungsi tersebut sebagai berikut:

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

$$(\boldsymbol{\lambda x.x^2 - 10)6 \rightarrow [6/x](x^2 - 10) = 36 - 10 = 26}$$

Dari penjelasan singkat ini, kita telah mengenal tentang **λ**-*calculus*. Di sini, kita melihat bahwa **λ**-*calculus* memiliki kemampuan untuk mengekspresikan fungsi secara *formal*.

3.2 Definisi dan Struktur λ-*Term*

Sub-bab ini memaparkan definisi-definisi yang berkaitan dengan **λ**-*term* (termasuk definisi **λ**-*term*) dan juga struktur **λ**-*term*.

3.2.1 Definisi λ-*Term*

Diberikan sejumlah tak berhingga *term-variables*. ***λ-term*** didefinisikan sebagai berikut:

- Semua *term-variables* adalah ***λ-term***. ***λ-term*** jenis ini disebut ***atom*** atau ***atomic term***.
- Jika $\boldsymbol{M}$ dan $\boldsymbol{N}$ adalah ***λ-term***, maka $(\boldsymbol{MN})$ adalah ***λ-term*** yang disebut dengan ***application***.
- Jika x adalah *term-variables* dan M adalah ***λ-term***, maka $(\boldsymbol{\lambda x.M})$ adalah ***λ-term*** yang disebut ***abstract*** atau ***λ-abstract***.

Berikut adalah beberapa contoh ***λ-term***:

- $\boldsymbol{x}$.
- $(\boldsymbol{xy})$.
- $\boldsymbol{\lambda x.y}$.
- $\boldsymbol{\lambda x.(xy)}$.
- $((\boldsymbol{\lambda x.(xy)})(\boldsymbol{\lambda z.(uv)}))$.
- $\boldsymbol{\lambda y.\lambda x.(xy)}$.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

3.2.2 Notasi Dalam $\boldsymbol{\lambda}$-*Term*

Berikut adalah konvensi notasi penulisan dalam menuliskan $\boldsymbol{\lambda}$-*term*:

- ***Term-variables*** dituliskan dengan huruf kecil u, v, w, x, y, z, dengan atau tanpa *sub-script* angka. Huruf yang berbeda menyatakan *variable* yang berbeda kecuali jika ada penjelasan lain yang telah dijelaskan sebelumnya.
- Sembarang $\boldsymbol{\lambda}$-*term* dituliskan dengan huruf besar L, M, N, P, Q, R, S, T dengan atau tanpa *sub-script* angka.
- "$\boldsymbol{\lambda}$-*term*" sering disebut juga sebagai ***"term"***.
- Tanda kurung dan simbol $\boldsymbol{\lambda}$ yang berturutan akan lebih sering dihilangkan. Contoh:
  - $\boldsymbol{\lambda uvw.M \equiv (\lambda u.(\lambda v.(\lambda w.M)))}$
  - $\boldsymbol{MNPQ \equiv (((MN)P)Q)}$

  Dalam hal ini aturan presedensi untuk sebuah aplikasi dari $\boldsymbol{\lambda}$-*term* adalah "*association to the left*". Sedangkan untuk sebuah abstraksi dari $\boldsymbol{\lambda}$-*term* adalah "*association to the right*".

3.2.3 *Length* (Panjang) $\boldsymbol{\lambda}$-*Term*

Length $|\boldsymbol{M}|$ untuk $\boldsymbol{\lambda}$-*term* M adalah jumlah kemunculan dari variables dalam M. Secara detail didefinisikan sebagai berikut:

- $|\boldsymbol{x}| = 1$
- $|\boldsymbol{MN}| = |\boldsymbol{M}| + |\boldsymbol{N}|$
- $|\boldsymbol{\lambda x.M}| = \mathbf{1} + |\boldsymbol{M}|$

Contoh:

- $|\boldsymbol{(\lambda z.x)(\lambda x.yx)}| = \mathbf{5}$
- $|\boldsymbol{(\lambda x.x)(\lambda xyz.x(yz))}| = \mathbf{8}$

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

- $|(xy)| = 2$

- $|\lambda xxxx.xx| = 6$

3.2.4 *Subterm* Dari λ-*term*

Subterm dari sebuah *term* M didefinisikan secara induksi pada M sebagai berikut:

- Jika M adalah sebuah *atom*, maka *subterm*-nya adalah dirinya sendiri (atom tersebut).

- Jika $M \equiv \lambda x.P$, maka *subterm*-nya adalah M itu sendiri dan semua *subterm* dari P.

- Jika $M \equiv P_1 P_2$, maka *subterm*-nya adalah M itu sendiri, semua *subterm* dari P_1, dan semua *subterm* dari P_2.

Contohnya adalah sebagai berikut:

- Jika $M \equiv (\lambda u.vw)$, maka *subterm*-nya adalah v, w, vw, dan $\lambda u.vw$.

3.2.5 Posisi Dalam λ-*Term*

Setiap kemunculan suatu *subterm* dalam λ-*term* memiliki suatu posisi. Adapun definisi dari posisi tersebut adalah sebagai berikut: Suatu posisi $p = i_1 \ldots i_m$ adalah barisan sejumlah simbol yang berhingga (mungkin kosong) sedemikian sehingga $i_1, \ldots, i_{m-1}$ adalah bulangan bulat dan i_m adalah bilangan bulat atau asterik (*). Panjang dari posisi ini adalah m. Jika m = 0, maka kita sebut p = $\emptyset$.

Jika m $\geq$ 1 dan i_m = 1, kita sebut p sebagai posisi ***fungsi***.

Jika m $\geq$ 1 dan i_m = 2, kita sebut p sebagai posisi ***argumen***.

Jika m $\geq$ 1 dan i_m = 0, kita sebut p sebagai posisi ***body***.

Jika m $\geq$ 1 dan i_m = *, kita sebut p sebagai posisi ***abstractor***.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

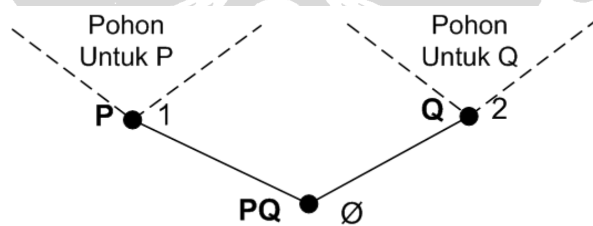
3.2.6 Pohon Konstruksi Dari λ-*term*

Kita dapat menampilkan struktur dari *term* sebagai sebuah pohon konstruksi. Dalam pohon konstruksi tersebut, setiap *node* memiliki 2 label, yaitu sebuah posisi *term* dan *sub-term* yang muncul pada posisi tersebut. Pohon ini didefinisikan untuk sembarang *term* $\boldsymbol{M}$ sebagai berikut:

- Jika $\boldsymbol{M} \equiv \boldsymbol{x}$, maka pohonnya adalah sebuah *node* tunggal yang memiliki label x dan posisi kosong. Jika digambarkan akan tampak seperti Gambar 3.1.

Gambar 3.1: Pembentukan pohon konstruksi untuk sebuah *term-variable*.

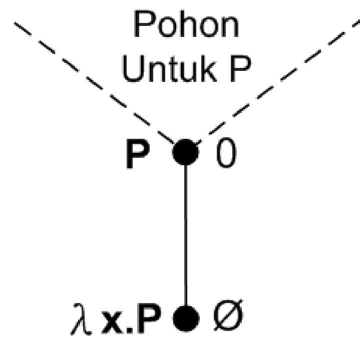
- Jika $\boldsymbol{M} \equiv \boldsymbol{PQ}$, maka pohonnya diperoleh dengan menambahkan angka "1" pada sisi paling kiri dari setiap label posisi di dalam pohon dari $\boldsymbol{P}$. Kemudian tambahkan angka "2" pada sisi paling kiri dari setiap label posisi di dalam pohon dari $\boldsymbol{Q}$. Lalu buat *node* baru dibawah dari dua pohon tersebut. Jika digambarkan akan tampak seperti Gambar 3.2.



Gambar 3.2: Pembentukan pohon konstruksi untuk sebuah aplikasi.

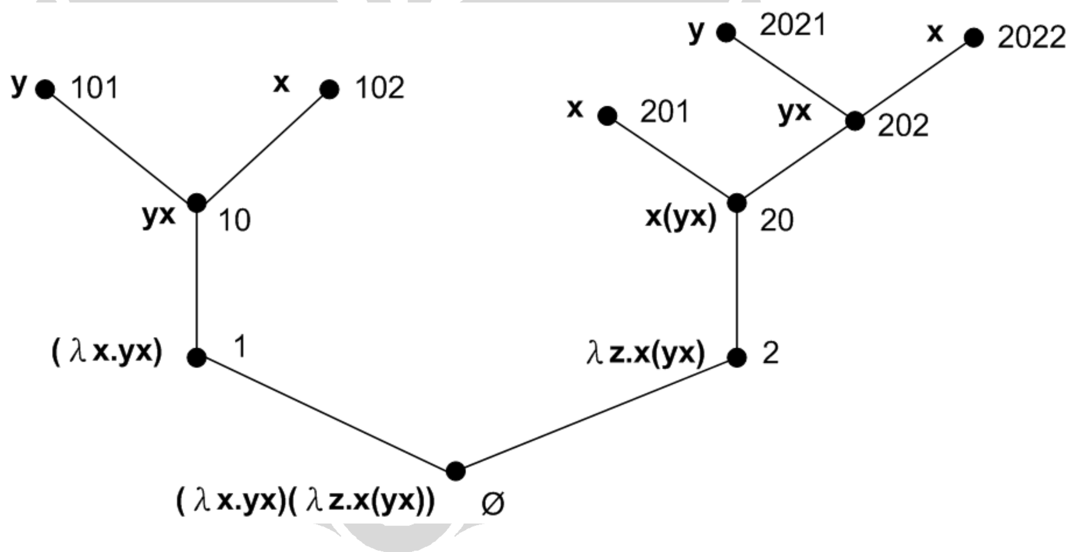
- Jika $\boldsymbol{M} \equiv \boldsymbol{\lambda x.P}$, maka pohonnya diperoleh dengan menambahkan angka "0" pada sisi paling kiri dari setiap label posisi di dalam pohon dari $\boldsymbol{P}$. Lalu buat *node* baru dibawah dari pohon tersebut. Jika digambarkan akan tampak seperti Gambar 3.3.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008



Gambar 3.3: Pembentukan pohon konstruksi untuk sebuah abstraksi.

Pada Gambar 3.4 ditunjukkan contoh sebuah pohon konstruksi yang menggambarkan konstruksi *term* $(\boldsymbol{\lambda x.yx})(\boldsymbol{\lambda z.x(yx)})$ dari *subterm-subterm*-nya dan juga beserta dengan posisi dari setiap *subterm* tersebut.



Gambar 3.4: *Construction-Tree* dari $(\boldsymbol{\lambda x.yx})(\boldsymbol{\lambda z.x(yx)})$.

3.2.7 Definisi *Body, Scope* dan *Covering Abstractor* dalam $\boldsymbol{\lambda}$-*Term*

Kemunculan $\boldsymbol{\lambda x}$ disebut sebagai ***abstractor***, dan kemunculan $\boldsymbol{x}$ didalamnya disebut sebagai ***binding occurrence*** dari $\boldsymbol{x}$. Semua kemunculan (*occurrence*) dari ***term*** dalam $\boldsymbol{M}$, selain *variable* yang termasuk *binding occurrences* disebut ***komponen*** dari $\boldsymbol{M}$.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

Kemudian misalkan ***λx.P*** adalah komponen dari ***term M***, komponen ***P*** disebut sebagai ***body*** dari ***λx.P*** atau ***scope*** dari ***abstractor λx***.

Covering abstractor dari komponen ***R*** pada ***M*** adalah *abstractor* pada ***M*** yang memiliki *scope* yang mengandung ***R***.

3.2.8 *Free* dan *Bound Variable* dalam λ-*Term*

Sebuah *variable* ***x*** yang kemunculannya tidak terikat (*non-binding occurrence*) dalam term ***M*** disebut ***bound*** dalam ***M*** jika dan hanya jika *variable* tersebut berada dalam *scope* dari kemunculan **λ***x* dalam ***M***. Selain pada kondisi tersebut, ***x*** disebut ***free*** di ***M***.

Sebuah *variable* ***x*** disebut ***bound*** di dalam ***M*** jika dan hanya jika ***M*** memiliki kemunculan **λ***x*. Kemudian ***x*** disebut ***free*** di dalam ***M*** jika dan hanya jika ***M*** mengandung *free occurrence* dari *x*.

Himpunan dari semua *free variables* di ***M*** dituliskan dengan notasi FV(M), dan Himpunan dari semua *bound variables* di ***M*** dituliskan dengan notasi BV(M).

Himpunan semua *free variables* dari **λ**-*term* dapat juga didefinisikan sebagai berikut [Bar92]:

- FV(x) = {x}
- FV(MN) = FV(M) ∪ FV(N)
- FV(**λ**x.M) = FV(M) - {x}

Dengan cara yang sama dan sedikit modifikasi, kita juga bisa mendefinisikan himpunan semua *bound variables* dari ***term M*** sebagai berikut:

- BV(x) = { }
- BV(MN) = BV(M) ∪ BV(N)
- BV(**λ**x.M) = BV(M) ∪ {x}

Sebuah *variable* ***x*** dapat berupa atau memiliki status sebagai *free variable* dan *bound variable* dalam suatu *term* ***M***. Misalnya pada *term* $\boldsymbol{M \equiv x(\lambda x.x)}$. Namun, kemunculan ***x*** dalam ***M*** tidak bisa sekaligus *free* dan *bound*. Satu hal lagi, ***x***

disebut *bound* juga pada $\boldsymbol{\lambda x.y}$ walaupun kemunculannya hanyalah *binding occurrence* (kemunculan terikat).

3.2.9 *Bound-Variable Clash*

Sebuah *term* $\boldsymbol{M}$ memiliki ***Bound-variable clash*** jika dan hanya jika $\boldsymbol{M}$ memiliki sebuah *abstractor* $\underline{\boldsymbol{\lambda} x}$, dan sebuah kemunculan $\boldsymbol{x}$ (baik *free* maupun *bound*) yang tidak berada pada *scope* abstractor tersebut. Contoh:

- $(\boldsymbol{\lambda z.M})(\boldsymbol{\lambda z.N})$
- $\boldsymbol{\lambda x.\lambda z.\lambda x.M}$
- $\boldsymbol{y(\lambda y.N)}$

3.2.10 *Closed Term (Combinator)*

Sebuah *term* disebut *Closed Term* atau *Combinator* jika *term* tersebut tidak memiliki *free-variable.* Contohnya adalah sebagai berikut:

- $\boldsymbol{B \equiv \lambda xyz.x(yz)}$
- $\boldsymbol{C \equiv \lambda xyz.xzy}$
- $\boldsymbol{K \equiv \lambda xy.x}$
- $\boldsymbol{W \equiv \lambda xy.xyy}$
- $\boldsymbol{\bar{O} \equiv \lambda xy.y}$
- $\boldsymbol{B' \equiv \lambda xyz.y(xz)}$
- $\boldsymbol{I \equiv \lambda x.x}$
- $\boldsymbol{S \equiv \lambda xyz.xz(yz)}$
- $\boldsymbol{Y \equiv \lambda x.(\lambda y.x(yy))(\lambda y.x(yy))}$ (Curry's *fixed-point combinator*)
- $\boldsymbol{\bar{I} \equiv \lambda xy.xy}$
- $\boldsymbol{\bar{n} \equiv \lambda xy.x^n y \equiv \lambda xy.x(x(\ldots(xy)\ldots))}$ ($\boldsymbol{n}$ $\boldsymbol{x}$ diaplikasikan pada $\boldsymbol{y}$)

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

3.3 Operasi Pada $\boldsymbol{\lambda}$-*Term*

Sub-bab ini memaparkan beberapa operasi yang berkaitan dengan $\boldsymbol{\lambda}$-*term*, yaitu Substitusi, $\boldsymbol{\alpha}$***-conversion***, $\boldsymbol{\beta}$*-reduction*, $\boldsymbol{\eta}$*-reduction*, dan $\boldsymbol{\beta\eta}$*-reduction.*

3.3.1 Substitusi

Substitusi terhadap *term*, dituliskan dengan $[\boldsymbol{N}/\boldsymbol{x}]\boldsymbol{M}$, merupakan suatu proses menggantikan setiap variable $\boldsymbol{x}$ yang muncul free (*free occurrence*) di $\boldsymbol{M}$ dengan N, dan juga mengubah beberapa *bound variable* yang diperlukan untuk mencegah variable yang free di $\boldsymbol{N}$ menjadi *bound* di $[\boldsymbol{N}/\boldsymbol{x}]\boldsymbol{M}$. Atau dengan kata lain substitusi bisa didefinisikan lebih *formal* sebagai berikut:

1. $[\boldsymbol{N}/\boldsymbol{x}]\boldsymbol{x} \equiv \boldsymbol{N}$
2. $[\boldsymbol{N}/\boldsymbol{x}]\boldsymbol{y} \equiv \boldsymbol{y}$
3. $[\boldsymbol{N}/\boldsymbol{x}](\boldsymbol{PQ}) \equiv (([\boldsymbol{N}/\boldsymbol{x}]\boldsymbol{P})([\boldsymbol{N}/\boldsymbol{x}]\boldsymbol{Q}))$
4. $[\boldsymbol{N}/\boldsymbol{x}](\boldsymbol{\lambda x.P}) \equiv \boldsymbol{\lambda x.P}$
5. $[\boldsymbol{N}/\boldsymbol{x}](\boldsymbol{\lambda y.P}) \equiv \boldsymbol{\lambda y.P}, \quad \textit{jika } \boldsymbol{x} \notin \boldsymbol{FV}(\boldsymbol{P})$
6. $[\boldsymbol{N}/\boldsymbol{x}](\boldsymbol{\lambda y.P}) \equiv \boldsymbol{\lambda y.}[\boldsymbol{N}/\boldsymbol{x}]\boldsymbol{P}, \quad \textit{jika } \boldsymbol{x} \in \boldsymbol{FV}(\boldsymbol{P}) \textit{ dan } \boldsymbol{y} \notin \boldsymbol{FV}(\boldsymbol{N})$
7. $[\boldsymbol{N}/\boldsymbol{x}](\boldsymbol{\lambda y.P}) \equiv \boldsymbol{\lambda z.}[\boldsymbol{N}/\boldsymbol{x}][\boldsymbol{z}/\boldsymbol{y}]\boldsymbol{P}, \quad \textit{jika } \boldsymbol{x} \in \boldsymbol{FV}(\boldsymbol{P}) \textit{ dan } \boldsymbol{y} \in \boldsymbol{FV}(\boldsymbol{N})$

3.3.2 α-*Conversion* (Pengubahan *Bound Variables*)

Sub-bab ini memaparkan pengertian dan definisi dari $\boldsymbol{\alpha}$-*conversion.* Adapun definisinya adalah sebagai berikut: Misalkan $\boldsymbol{y} \notin \boldsymbol{FV}(\boldsymbol{M})$, kita bisa lakukan hal berikut:

$$\boldsymbol{\lambda x.M} \equiv_\alpha \boldsymbol{\lambda y.}[\boldsymbol{y}/\boldsymbol{x}]\boldsymbol{M}$$

dan proses penggantian $\boldsymbol{\lambda x.M}$ menjadi $\boldsymbol{\lambda y.}[\boldsymbol{y}/\boldsymbol{x}]\boldsymbol{M}$ disebut ***change of bound variables*** (penggantian variabel terikat). Jika $\boldsymbol{P}$ berubah menjadi $\boldsymbol{Q}$ melalui sejumlah langkah pengubahan variabel terikat yang berhingga (boleh juga tanpa satu langkah pun), maka kita bisa katakan $\boldsymbol{P}$ $\boldsymbol{\alpha}$***-converts*** ke $\boldsymbol{Q}$, atau

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

$$P \equiv_\alpha Q.$$

Berikut adalah beberapa ***Lemma*** tentang ***α-conversion***:

- $P \equiv_\alpha Q \implies |P| = |Q|$.
- $P \equiv_\alpha Q \implies FV(P) = FV(Q)$.
- Semua *term* dapat diterapkan α-*conversion* menjadi *term* lain tanpa mengalami *bound-variable clashes.*

3.3.3 Reduksi β (*β-reduction*)

Reduksi β atau β-*reduction* merupakan suatu prosedur penulisan ulang suatu *term* (*term-rewriting procedure*). Sub-bab ini, membahas pengertian prosedur β-*reduction* beserta beberapa definisi dan teorema yang berkaitan.

3.3.3.1 Definisi *β-redex*, *β-contraction*, *β-contracts*, *contractum*

Sebuah ***β-redex*** adalah sembarang *term* $(\lambda x.M)N$. ***Contractum***-nya adalah $[N/x]M$, dan aturan penulisan ulangnya (***re-write rule***) adalah sebagai berikut:

$$(\lambda x.M)N \;\rhd_{1\beta}\; [N/x]M$$

Kita sebut P ***β-contracts*** kepada Q $(P \rhd_{1\beta} Q)$ jika dan hanya jika P mengandung kemunculan β-*redex* $R \equiv (\lambda x.M)N$ dan Q sebagai hasil dari penggantian β-*redex* dengan $[N/x]M$. Kemudian kita sebut *triple* $\langle P, R, Q\rangle$ sebagai ***β-contraction*** dari P.

Contoh:

$(\lambda uv.u)uv \;\rhd_{1\beta}\; (\lambda v.u)v \;\rhd_{1\beta}\; u$

$\langle(\lambda uv.u)uv, (\lambda uv.u)u, (\lambda v.u)v\rangle$ adalah β-*contraction* dari $(\lambda uv.u)uv$.

3.3.3.2 Definisi *β-reduction*

β-reduction dari sebuah *term* P adalah barisan β-*contraction* (berhingga atau tidak) yang memiliki bentuk:

$$\langle P_1, R_1, Q_1\rangle, \langle P_2, R_2, Q_2\rangle$$

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

Dimana $\boldsymbol{P_1} \equiv_\alpha \boldsymbol{P}$ dan $\boldsymbol{Q_i} \equiv_\alpha \boldsymbol{P_{i+1}}$ untuk i = 1,2,... (barisan tersebut juga boleh kosong). Kita sebut sebuah reduksi dari $\boldsymbol{P}$ ke $\boldsymbol{Q}$ berhingga jika dan hanya jika reduksi tersebut memiliki *β-contraction* sejumlah n $\geq$ 1 dan $\boldsymbol{Q_n} \equiv_\alpha \boldsymbol{Q}$ atau reduksi tersebut tidak memiliki *β-contraction* sama sekali dan $\boldsymbol{P} \equiv_\alpha \boldsymbol{Q}$. Sebuah reduksi dari $\boldsymbol{P}$ ke $\boldsymbol{Q}$ disebut berhenti atau berakhir pada $\boldsymbol{Q}$ jika dan hanya jika ada reduksi dari $\boldsymbol{P}$ ke Q, kita sebut $\boldsymbol{P}$ ***β-reduces*** ke $\boldsymbol{Q}$, atau $\boldsymbol{P} \triangleright_\beta \boldsymbol{Q}$.

3.3.3.3 Definisi *β-conversion*, *β-equal* dan *β-expansion*

$\boldsymbol{P}$ disebut ***β-converts*** ke $\boldsymbol{Q}$ atau $\boldsymbol{P}$ adalah ***β-equal*** dengan $\boldsymbol{Q}$ atau

$$\boldsymbol{P} =_\beta \boldsymbol{Q}$$

jika dan hanya jika kita dapat mengubah $\boldsymbol{P}$ menjadi $\boldsymbol{Q}$ dengan sejumlah barisan *β-reduction* dan *operasi kebalikan β-reduction. Operasi* kebalikan *β-reduction* disebut juga dengan ***β-expansion***.

3.3.3.4 *β-normal form*

Sebuah ***β-normal form*** adalah *term* yang tidak memiliki *β-redex.* Kelas dari semua *β-normal form* disebut ***β-nf***, dan kita sebut *term* $\boldsymbol{M}$ memiliki ***β-nf*** $\boldsymbol{N}$ jika dan hanya jika

$$\boldsymbol{M} \triangleright_\beta \boldsymbol{N} \text{ dan } \boldsymbol{N} \in \boldsymbol{\beta}\text{-}\boldsymbol{nf}.$$

Contoh:

Untuk $(\boldsymbol{\lambda yx.y})\boldsymbol{yx} \triangleright_\beta \boldsymbol{y}$,

$\boldsymbol{y}$ adalah $\boldsymbol{\beta}$-nf dari $(\boldsymbol{\lambda yx.y})\boldsymbol{yx}$

Secara informal, proses reduksi dapat dikatakan sebagai komputasi, dan *β-normal form* (β-nf) sebagai hasilnya.

3.3.3.5 Struktur *β-normal form*

Setiap *β-nf* N dapat diekspresikan secara unik dalam bentuk sebagai berikut:

$$\boldsymbol{N} \equiv \boldsymbol{\lambda x_1 \ldots x_m . y N_1 \ldots N_n} \qquad (\boldsymbol{m \geq 0,\ n \geq 0})$$

dengan $\boldsymbol{N_1, \ldots, N_n}$ adalah *β-nf* dan jika $\boldsymbol{N}$ adalah *closed-term*, maka $\boldsymbol{y} \in \{\boldsymbol{x_1, \ldots, x_m}\}$.

3.3.4 Reduksi η (η-*reduction*)

Sub-bab ini membahas $\boldsymbol{\eta}$-*reduction*, salah satu bentuk reduksi pada $\boldsymbol{\lambda}$-*term* selain $\boldsymbol{\beta}$-*reduction*.

3.3.4.1 Definisi η-*reduction*

Sebuah $\boldsymbol{\eta}$-*redex* adalah sembarang *term* $\boldsymbol{\lambda x.Mx}$ dengan $\boldsymbol{x \notin FV(M)}$, dan aturan penulisan ulangnya (*re-write rule*) adalah

$$\boldsymbol{\lambda x.Mx} \;\triangleright_{1\eta}\; \boldsymbol{M}$$

dengan $\boldsymbol{M}$ sebagai ***contractum***-nya. Untuk definisi $\boldsymbol{\eta}$-*contracts*, $\boldsymbol{\eta}$-*reduce* ($\triangleright_{\boldsymbol{\eta}}$), $\boldsymbol{\eta}$-*converts* ($=_{\boldsymbol{\eta}}$), dll, didefinisikan mirip dengan definisi pada konsep $\boldsymbol{\beta}$-*reduction*.

Contoh $\boldsymbol{\eta}$-*reduction*: $\boldsymbol{\lambda x.zx} \;\triangleright_{1\beta}\; \boldsymbol{z}$

Ekspresi $\boldsymbol{\lambda x.Mx}$ dan $\boldsymbol{M}$ memiliki perilaku komputasi yang sama pada semua argumen, karena

$$\boldsymbol{(\lambda x.Mx)y} \;\triangleright_{1\beta}\; \boldsymbol{My}$$

untuk sembarang y. Oleh karena itu, aturan $\boldsymbol{\eta}$-*reduction* ini dapat dikatakan mengidentifikasi *term* tertentu yang mewakili suatu fungsi yang memiliki perilaku yang sama, yang direpresentasikan dengan cara yang berbeda.

3.3.4.2 Definisi η-*family*

$\boldsymbol{\eta}$-*family* $\boldsymbol{\{P\}_\eta}$ dari suatu *term* $\boldsymbol{P}$ adalah himpunan dari semua *term* $\boldsymbol{Q}$ sedemikian sehingga $\boldsymbol{P} \;\triangleright_\eta\; \boldsymbol{Q}$.

3.3.4.3 η-*normal form*

Sebuah ***η-normal form*** adalah *term* yang tidak memiliki $\boldsymbol{\eta}$-*redex* sama sekali. Kelas dari semua $\boldsymbol{\eta}$-*normal form* disebut $\boldsymbol{\eta}$-***nf***, dan kita sebut *term* $\boldsymbol{M}$ memiliki $\boldsymbol{\eta}$-***nf*** $\boldsymbol{N}$ jika dan hanya jika

$$\boldsymbol{M} \;\triangleright_\eta\; \boldsymbol{N} \text{ dan } \boldsymbol{N} \in \boldsymbol{\eta}\text{-}\boldsymbol{nf}.$$

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

3.3.5 Reduksi $\beta\eta$ (*$\beta\eta$-reduction*)

Sub-bab ini membahas ***$\beta\eta$**-reduction*. Reduksi ini merupakan gabungan konsep ***β**-reduction* dan ***η**-reduction*.

3.3.5.1 Definisi *$\beta\eta$-reduction*

Sebuah ***$\beta\eta$-redex*** adalah sembarang **β-** atau ***η-redex***. Sedangkan definisi untuk ***$\beta\eta$**-contracts*, ***$\beta\eta$**-reduce* ($\triangleright_{\beta\eta}$), ***$\beta\eta$**-converts* ($=_{\beta\eta}$) didefinisikan mirip dengan definisi pada konsep ***β**-reduction*.

3.3.5.2 *$\beta\eta$-normal form*

Sebuah ***$\beta\eta$-normal form*** adalah *term* yang tidak memiliki ***$\beta\eta$**-redex* sama sekali. Kelas dari semua ***$\beta\eta$**-normal form* disebut ***$\beta\eta$-nf***, dan kita sebut *term* $\boldsymbol{M}$ memiliki ***$\beta\eta$-nf*** $\boldsymbol{N}$ jika dan hanya jika

$$\boldsymbol{M} \triangleright_{\beta\eta} \boldsymbol{N} \text{ dan } \boldsymbol{N} \in \beta\eta\text{-}\boldsymbol{nf}.$$

Bab 4

Type-Assignment pada $\boldsymbol{\lambda}$-*term* ($\text{TA}_{\boldsymbol{\lambda}}$)

Salah satu *type theory* yang paling sederhana adalah *Type Assignment* (TA). TA ini memiliki dua jenis, yaitu $\text{TA}_{\boldsymbol{C}}$ untuk *combinatory logic* dan $\text{TA}_{\boldsymbol{\lambda}}$ untuk $\boldsymbol{\lambda}$-*calculus*. Dalam $\text{TA}_{\boldsymbol{\lambda}}$, terdapat dua pendekatan utama, yaitu pendekatan yang dilakukan oleh Church dan Curry.

Bab ini menjelaskan *type-assignment* pada $\boldsymbol{\lambda}$-*calculus* ($\text{TA}_{\boldsymbol{\lambda}}$). Pada bab ini hanya dijelaskan pendekatan $\text{TA}_{\boldsymbol{\lambda}}$ yang dilakukan oleh Curry. Penjelasan pada bab ini dimulai dengan pemaparan mengenai pengertian (definisi) dari tipe, dan juga beberapa definisi terkait dengan tipe. Kemudian, bab ini dilanjutkan dengan penjelasan sistem $\text{TA}_{\boldsymbol{\lambda}}$. Tidak hanya berhenti di situ, beberapa contoh pemberian tipe pada $\boldsymbol{\lambda}$-*term* juga diberikan untuk memudahkan pemahaman. Acuan utama dalam bab ini adalah [Hin97], termasuk definisi-definisi dan teorema yang dijelaskan pada bab ini.

4.1 Pengertian *Type* (Tipe)

Sub-bab ini menjelaskan pengertian dari tipe. Penjelasan pada sub-bab ini dimulai dengan definisi dari tipe, dan dilanjutkan dengan beberapa definisi terkait dengan tipe.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

4.1.1 Definisi *Type* (Tipe)

Berikut adalah definisi dari tipe. Diberikan sejumlah ***type-variables*** yang berbeda dari *term-variable*, tipe dari suatu $\boldsymbol{\lambda}$-*term* didefinisikan sebagai ekspresi linguistik dengan struktur sebagai berikut:

- Setiap *type-variable* merupakan sebuah tipe (disebut ***atom***).
- Jika $\boldsymbol{\sigma}$ dan $\boldsymbol{\tau}$ adalah sebuah tipe, maka $(\boldsymbol{\sigma} \rightarrow \boldsymbol{\tau})$ merupakan sebuah tipe (disebut **tipe komposit** (***composite type***)).

Contoh dari tipe:

- $\boldsymbol{\sigma}$
- $(\boldsymbol{\sigma} \rightarrow \boldsymbol{\tau})$
- $(\boldsymbol{\rho} \rightarrow (\boldsymbol{\sigma} \rightarrow \boldsymbol{\tau}))$
- $((\boldsymbol{\rho} \rightarrow \boldsymbol{\sigma}) \rightarrow (\boldsymbol{\sigma} \rightarrow \boldsymbol{\tau}))$
- $((\boldsymbol{\rho} \rightarrow \boldsymbol{\sigma}) \rightarrow \boldsymbol{\tau})$

4.1.2 Notasi Penulisan Tipe

Berikut adalah notasi penulisan tipe:

- ***Type-variable***, dituliskan dengan huruf "a", "b", "c", "d", "e", "f", "g" dengan atau tanpa *subscript* angka. Huruf yang berbeda menyatakan *variable* yang berbeda kecuali jika ada penjelasan lain yang telah dijelaskan sebelumnya.
- Sembarang tipe (***arbitrary type***), dituliskan dengan huruf kecil Yunani kecuali "$\boldsymbol{\lambda}$".
- Tanda kurung terkadang sering dihilangkan dari penulisan sebuah tipe (namun tidak selalu). Misalnya:

$$\boldsymbol{\rho \rightarrow \sigma \rightarrow \tau \equiv (\rho \rightarrow (\sigma \rightarrow \tau))}$$

  Dalam hal ini aturan presedensi untuk sebuah tipe adalah "*association to the right*".

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

4.1.3 Beberapa Definisi Seputar Tipe

Length (panjang) dari sebuah tipe $\boldsymbol{\tau}$ atau $|\boldsymbol{\tau}|$ adalah jumlah kemunculan *type-variable* dalam tipe $\boldsymbol{\tau}$ tersebut. Lebih formal lagi didefinisikan sebagai berikut:

- $|\boldsymbol{a}| \equiv 1$.
- $|\boldsymbol{\rho} \rightarrow \boldsymbol{\sigma}| \equiv |\boldsymbol{\rho}| + |\boldsymbol{\sigma}|$.

Jumlah *type-variable* yang berbeda yang muncul dalam $\boldsymbol{\tau}$ disebut:

$$\|\boldsymbol{\tau}\|.$$

Kemudian himpunan semua *type-variable* dalam $\boldsymbol{\tau}$ disebut:

$$\boldsymbol{Vars(\tau)},$$

dan himpunan semua *type-variable* pada suatu barisan tipe $\langle\boldsymbol{\tau_1, \ldots, \tau_n}\rangle$ disebut:

$$\boldsymbol{Vars(\tau_1, \ldots, \tau_n)}.$$

Contoh:

$\boldsymbol{\tau \equiv (a \rightarrow b \rightarrow c) \rightarrow (f \rightarrow b \rightarrow d)}$
$\boldsymbol{|\tau| = 6}$, $\boldsymbol{\|\tau\| = 5}$, $\boldsymbol{Vars(\tau) = \{a, b, c, d, f\}}$.

4.2 Sistem $\text{TA}_{\boldsymbol{\lambda}}$

Sub-bab ini menjelaskan sistem $\text{TA}_{\boldsymbol{\lambda}}$. $\text{TA}_{\boldsymbol{\lambda}}$ memiliki sejumlah tak berhingga aksioma dan dua aturan deduksi yang dijelaskan pada sub-bab 4.2.4. Sebelum masuk ke bagian tersebut, ada beberapa definisi yang perlu diketahui berkaitan dengan sistem $\text{TA}_{\boldsymbol{\lambda}}$, yaitu definisi *type-assignment*, *type-context*, dan *$TA_{\boldsymbol{\lambda}}$-formula.*

4.2.1 Definisi *Type-Assignment*

Type-assignment merupakan sebuah ekspresi

$$\boldsymbol{M : \tau}$$

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

dimana M adalah λ-*term* dan τ adalah sebuah tipe. Kita sebut M sebagai ***subject*** **(subjek)** dan τ sebagai ***predicate*** **(predikat)**. "$M{:}\tau$" dibaca "memberikan M tipe τ", atau "M memiliki tipe τ", atau "M menunjukkan anggota dari sembarang himpunan yang dinyatakan oleh τ".

4.2.2 Definisi *Type-Context*

Type-Context (Γ) adalah himpunan *type-assignment* yang berhingga (mungkin kosong)

$$\Gamma = \{x_1 : \rho_1, \ldots, x_m : \rho_m\}$$

dimana subjeknya adalah *term-variable* yang ***monovalent*** atau ***consistent***, dalam artian tidak ada variable yang menjadi subjek untuk lebih dari satu *type-assignment*.

Berikut ini adalah beberapa definisi yang berkaitan dengan *type-context* (Γ):

- Himpunan semua subjek dari sebuah Γ dinyatakan sebagai berikut:

  $$\textit{Subjects}(\Gamma) = \{x_1, \ldots, x_m\}$$

- Hasil dari menghilangkan *type-assignment* yang memiliki subjek x dalam Γ (jika Γ memilikinya) dituliskan sebagai berikut:

  $$\Gamma - x$$

  Jika $x \notin \textit{Subject}(\Gamma)$, maka kita definisikan $\Gamma - x = \Gamma$.

- Hasil dari menghilangkan semua *type-assignment* $x_i : \rho_i$, dimana $x_i \notin FV(M)$ (M adalah *term* yang diberikan) disebut:

  $$\Gamma \upharpoonright M$$

  atau "Γ terbatas pada M" ("Γ *restricted to* M").

- Γ disebut sebagai "***M-context***" (untuk suatu *term* M) jika dan hanya jika $\textit{Subject}(\Gamma) = FV(M)$.

- Γ_1 ***konsisten*** dengan Γ_2 jika dan hanya jika $\Gamma_1 \cup \Gamma_2$ ***konsisten***.

- $\Gamma_1, \ldots, \Gamma_n$ disebut ***saling konsisten*** (***mutually consistent***) jika dan hanya jika *union* dari mereka semua konsisten.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

4.2.3 Definisi Formula TA_λ

Untuk sembarang Γ, M dan τ, *tripe* (Γ, M, τ) disebut sebagai formula TA_λ dan dituliskan sebagai berikut:

$$\Gamma \mapsto M : \tau,$$

atau hanya $\mapsto M : \tau$ jika Γ kosong ($\Gamma = \{\}$). Dalam hal ini, kita sebut M adalah subjek dari formula ini dan τ adalah predikatnya.

4.2.3.1 Notasi Penulisan Formula TA_λ

Dalam penulisan, terkadang dilakukan penyingkatan sebagai berikut:

- $\{ x_1 : \sigma_1, \ldots, x_n : \sigma_n \} \mapsto M : \tau$, disingkat menjadi

  $x_1 : \sigma_1, \ldots, x_n : \sigma_n \mapsto M : \tau$.

- $\Gamma \cup \{ y_1 : \sigma_1, \ldots, y_n : \sigma_n \} \mapsto M : \tau$, disingkat menjadi

  $\Gamma, y_1 : \sigma_1, \ldots, y_n : \sigma_n \mapsto M : \tau$.

4.2.4 Aksioma dan Aturan Deduksi dalam TA_λ

Aturan deduksi merupakan sekumpulan aturan formal untuk melakukan deduksi dalam TA_λ. TA_λ memiliki sejumlah tak berhingga aksioma dan dua aturan deduksi yang disebut:

- ($\rightarrow$**E**) atau $\rightarrow$-***elimination*** (*arrow-elimination*)
- ($\rightarrow$**I**) atau $\rightarrow$-***introduction*** (*arrow-introduction*)

Aksioma dan aturan deduksi ini dijelaskan pada sub-bab 4.2.4.1 dan 4.2.4.2.

4.2.4.1 Aksioma dalam TA_λ

Untuk setiap *term-variable* x dan setiap tipe τ, TA_λ memiliki aksioma:

$$x : \tau \mapsto x : \tau$$

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

4.2.4.2 Aturan Deduksi dalam TA_λ

Aturan deduksi dari TA_λ adalah:

$$\frac{\Gamma_1 \mapsto P : (\sigma \to \tau) \qquad \Gamma_2 \mapsto Q : \sigma}{\Gamma_1 \cup \Gamma_2 \mapsto (PQ) : \tau} (\to E) \qquad [Jika\ \ \Gamma_1 \cup \Gamma_2\ \ konsisten]$$

$$\frac{\Gamma \mapsto P : \tau}{\Gamma - x \mapsto (\lambda x.P) : (\sigma \to \tau)} (\to I) \qquad [Jika\ \ \Gamma\ \ konsisten\ dengan\ \ x : \sigma]$$

Maksud kondisi pada $(\to I)$, yaitu Γ konsisten dengan $x : \sigma$, adalah Γ mengandung $x : \sigma$ atau Γ tidak mengandung *type-assignment* yang memiliki subject x sama sekali. Untuk aturan $(\to I)$ dibagi lagi menjadi dua kasus:

- *Discharge* x dari Γ dimana $\Gamma = \Gamma_1 \cup \{x\}$:

$$\frac{\Gamma_1,\ x : \sigma \mapsto P : \tau}{\Gamma_1 \mapsto (\lambda x.P) : (\sigma \to \tau)} (\to I)_{main} \qquad [Jika\ \ x\ \notin\ Subjects(\Gamma_1)]$$

- *Discharge* x *vacuously*:

$$\frac{\Gamma_1 \mapsto P : \tau}{\Gamma_1 \mapsto (\lambda x.P) : (\sigma \to \tau)} (\to I)_{vacuous} \qquad [Jika\ \ x\ \notin\ Subjects(\Gamma_1)]$$

4.2.5 Deduksi TA_λ (∇)

Sebuah deduksi TA_λ ∇ adalah struktur *tree* dari formula TA_λ, yang pada ujung atas cabangnya adalah aksioma, dan dibawahnya adalah hasil deduksi dari formula TA_λ yang berada di atasnya dengan menerapkan aturan deduksi yang telah didefinisikan pada 4.2.4.2. Formula yang berada di posisi paling bawah dari ∇ disebut ***conclusion*** (kesimpulan). Jika formula tersebut adalah $\Gamma \mapsto M : \tau$ kita sebut ∇ sebagai deduksi dari $\Gamma \mapsto M : \tau$ atau deduksi $M : \tau$ dari Γ. Kemudian kita sebut $\Gamma \mapsto M : \tau$ sebagai ***TA_λ-deductible***. Pada kasus khusus dimana $\Gamma = \emptyset$, ∇ disebut bukti (*proof*) dari *assignment* $M : \tau$.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

4.2.5.1 Contoh Deduksi $\mathbf{TA}_{\boldsymbol{\lambda}}$ (∇)

Berikut ini adalah pemaparan beberapa contoh proses deduksi TA_{λ}:

- Misalkan I $\equiv$ $\boldsymbol{\lambda x.x}$, berikut adalah deduksi untuk $\mapsto \boldsymbol{I : a \to a}$

$$\dfrac{x : a \mapsto x : a}{\mapsto (\lambda x.x) : a \to a}\,(\to I)_{main}$$

- Misalkan K $\equiv$ $\boldsymbol{\lambda xy.x}$, deduksi

  $\mapsto \boldsymbol{K : a \to b \to a}$

  adalah sebagai berikut:

$$\dfrac{\dfrac{x : a \mapsto x : a}{x : a \mapsto (\lambda y.x) : b \to a}\,(\to I)_{vac}}{\mapsto (\lambda xy.x) : a \to b \to a}\,(\to I)_{main}$$

- Misalkan II $\equiv$ $\boldsymbol{(\lambda x.x)(\lambda x.x)}$, deduksi

  $\mapsto \boldsymbol{II : a \to a}$

  adalah sebagai berikut:

$$\dfrac{\dfrac{x : (a \to a) \mapsto x : (a \to a)}{\mapsto (\lambda x.x) : (a \to a) \to a \to a}\,(\to I) \qquad \dfrac{x : a \mapsto x : a}{\mapsto (\lambda x.x) : a \to a}\,(\to I)}{\mapsto (\lambda x.x)(\lambda x.x) : a \to a}\,(\to E)$$

- Misalkan B $\equiv$ $\boldsymbol{\lambda xyz.x(yz)}$, deduksi

  $\mapsto \boldsymbol{B : (a \to b) \to (c \to a) \to c \to b}$

  adalah sebagai berikut:

$$\dfrac{\dfrac{\dfrac{\dfrac{x : a \to b \mapsto x : a \to b \qquad \dfrac{y : c \to a \mapsto y : c \to a \qquad z : c \mapsto z : c}{y : c \to a,\ z : c \mapsto yz : a}\,(\to E)}{x : a \to b,\ y : c \to a,\ z : c \mapsto (x(yz)) : b}\,(\to E)}{x : a \to b,\ y : c \to a\ \mapsto \lambda z.(x(yz)) : c \to b}\,(\to I)_{main}}{x : a \to b,\ \mapsto \lambda yz.(x(yz)) : (c \to a) \to c \to b}\,(\to I)_{main}}{\mapsto \lambda xyz.(x(yz)) : (a \to b) \to (c \to a) \to c \to b}\,(\to I)_{main}$$

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

4.2.6 *Deducibility* dalam TA$_\lambda$

Misalkan $\boldsymbol{\Gamma}$ adalah sebuah *type-context*. Kita sebut

$$\boldsymbol{\Gamma} \ \vdash_\lambda \boldsymbol{M : \tau}$$

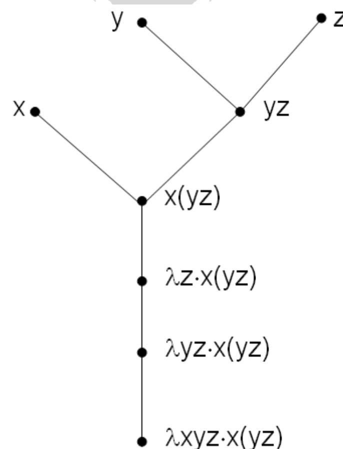
jika dan hanya jika terdapat deduksi TA$_\lambda$ dari formula $\boldsymbol{\Gamma' \mapsto M : \tau}$ untuk suatu $\boldsymbol{\Gamma' \subseteq \Gamma}$. Pada kasus khusus dimana $\boldsymbol{\Gamma = \emptyset}$, $\boldsymbol{\Gamma} \ \vdash_\lambda \boldsymbol{M : \tau}$ dapat kita tuliskan sebagai berikut:

$$\vdash_\lambda \boldsymbol{M : \tau}$$

dan $\vdash_\lambda \boldsymbol{M : \tau}$ dapat dibaca sebagai "M memiliki tipe $\boldsymbol{\tau}$ dalam TA$_\lambda$" atau "$\boldsymbol{\tau}$ adalah tipe dari M dalam TA$_\lambda$".

4.3 Teorema Konstruksi Subjek (*Subject Construction Theorem*) dalam TA$_\lambda$

Satu sifat yang sangat penting dari deduksi di TA$_\lambda$ adalah struktur *tree* dari deduksi $\boldsymbol{\Gamma \mapsto M : \tau}$ yang sama persis dengan struktur dari $\boldsymbol{M}$. Misalnya untuk $\boldsymbol{B \equiv \lambda xyz.x(yz)}$ (deduksi untuk *term* ini ditunjukkan pada 4.2.5.1), jika kita menghapus semua bagian dari setiap formula dalam deduksi, kecuali subjek dari formula tersebut, maka dihasilkan *construction-tree* seperti yang ditunjukkan pada gambar 4.1:



Gambar 4.1: *Construction-Tree* dari $\boldsymbol{\lambda xyz.x(yz)}$.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

Teorema berikut ini mendefinisikan hubungan antara deduksi dan *term* (seperti yang telah kita lihat) secara lebih formal. Teorema ini disebut ***Subject-Construction Theorem***:

Misalkan ∇ adalah deduksi TA_{λ} dari formula $\Gamma \mapsto M : \tau$,

1. Jika kita menghapus semua bagian dalam setiap formula di ∇, terkecuali setiap subjek dari tiap *assignment*-nya, maka ∇ akan berubah menjadi *tree* dari *term* yang sama persis dengan *construction-tree* untuk M.

2. Jika M adalah sebuah *atom*, dan misalkan $M \equiv x$, maka $\Gamma = \{x : \tau\}$ dan ∇ hanya memiliki satu formula yaitu aksioma

$$x : \tau \mapsto x : \tau$$

3. Jika $M \equiv PQ$, maka langkah terakhir dalam ∇ pastilah merupakan penerapan aturan $(\rightarrow E)$ pada dua formula yang memiliki bentuk

$$\Gamma \upharpoonright P \mapsto P : \sigma \rightarrow \tau, \quad \Gamma \upharpoonright Q \mapsto Q : \sigma$$

untuk suatu σ

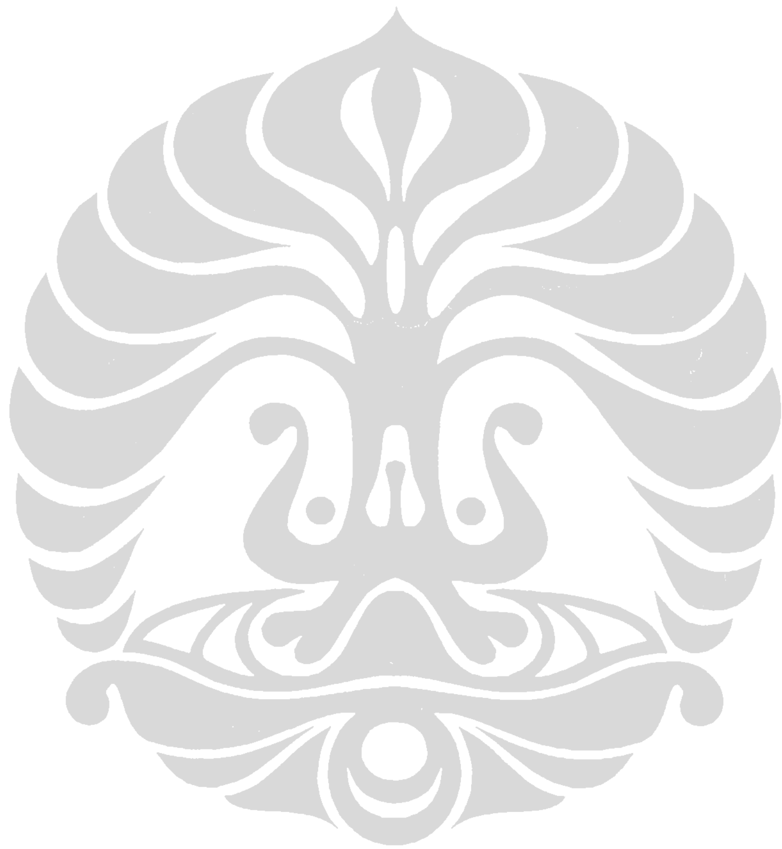
4. Jika $M \equiv \lambda x.P$ maka τ pasti memiliki bentuk $\rho \rightarrow \sigma$. Kemudian jika $x \in FV(P)$, maka langkah terakhir dari ∇ pastilah merupakan penerapan aturan $(\rightarrow I)_{main}$ pada formula yang memiliki bentuk

$$\Gamma, x : \rho \mapsto P : \sigma,$$

dan jika $x \notin FV(P)$, maka langkah terakhir dari ∇ pastilah merupakan penerapan aturan $(\rightarrow I)_{vac}$ pada formula yang memiliki bentuk

$$\Gamma \mapsto P : \sigma,$$

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008



Bab 5

Principal-Type (*PT*) *Algorithm*: Algoritma Pemberian Tipe pada $\boldsymbol{\lambda}$-*term*

Menurut Teorema *Principal type* (PT), "Setiap *typable term* memiliki *principal deduction* dan *principal type* dalam $\mathrm{TA}_{\boldsymbol{\lambda}}$. Lebih jauh lagi, terdapat algoritma yang dapat menentukan apakah suatu $\boldsymbol{\lambda}$-*term* M dapat memiliki tipe dalam $\mathrm{TA}_{\boldsymbol{\lambda}}$, dan jika jawabannya adalah 'ya', maka algoritma tersebut akan mengeluarkan *principal deduction* dan *principal type* dari $\boldsymbol{M}$". Algoritma tersebut dinamakan algoritma *Principal-type*, atau dikenal juga dengan nama *Type-checking algorithm.* Algoritma PT merupakan suatu algoritma yang bisa menentukan suatu $\boldsymbol{\lambda}$-*term* dapat diberikan tipe atau tidak. Jika $\boldsymbol{\lambda}$-*term* tersebut dapat diberikan tipe, maka algoritma ini akan mengeluarkan *principal type* (Tipe yang paling umum yang dapat diterima suatu *term* dalam $\mathrm{TA}_{\boldsymbol{\lambda}}$) dari $\boldsymbol{\lambda}$-*term* tersebut. Jika $\boldsymbol{\lambda}$-*term* tersebut tidak dapat diberikan tipe, maka algoritma ini akan memberikan suatu pernyataan yang benar bahwa $\boldsymbol{\lambda}$-*term* tersebut tidak dapat diberikan tipe.

Bab ini menyajikan penjelasan algoritma PT. Penjelasan dimulai dengan pemaparan konsep-konsep yang mendukung untuk memahami algoritma PT, diantaranya adalah substitusi tipe, definisi *principal type*, *principal pair*, dan *principal deduction.* Kemudian bab ini juga menjelaskan teori unifikasi beserta algoritma unifikasi yang digunakan dalam algoritma *Principal-type.* Setelah itu, penjelasan dilanjutkan dengan

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

ide atau prinsip dasar untuk bagian inti dari algoritma *Principal-type*, dan terakhir barulah dijelaskan algoritma *Principal-type*. Acuan utama dalam bab ini adalah [Hin97], termasuk definisi-definisi, teorema dan lemma yang dijelaskan pada bab ini.

5.1 Substitusi Tipe

Sub-bab ini menjelaskan konsep substitusi tipe (*type-substitution*). Substitusi tipe merupakan salah satu konsep penting yang digunakan dalam algoritma PT.

5.1.1 Definisi Substitusi Tipe

Suatu substitusi tipe s adalah sebuah ekspresi

$$[\sigma_1/a_1, \ldots, \sigma_n/a_n]$$

dimana $a_1, \ldots, a_n$ adalah *type-variable* yang berbeda dan $\sigma_1, \ldots, \sigma_n$ adalah sembarang tipe. Untuk suatu τ, kita definisikan $s(\tau)$ sebagai tipe yang dihasilkan dari melakukan substitusi secara bersamaan untuk menggantikan a_1 dengan σ_1, ..., a_n dengan σ_n secara keseluruhan pada τ. Berikut ini adalah definisi yang lebih *formal*:

1. $s(a_i) \equiv \sigma_i$
2. $s(b) \equiv b$ [Jika b adalah atom $\notin \{a_1, \ldots, a_n\}$]
3. $s(\rho \to \sigma) \equiv s(\rho) \to s(\sigma)$

kita sebut $s(\tau)$ sebagai ***instance*** dari τ.

Berikut ini adalah contoh substitusi tipe:

1. Misalkan

   $s \equiv [(a \to b)/c]$, dan

   $\tau \equiv (b \to c) \to a \to c$,

   $s(\tau) \equiv (b \to a \to b) \to a \to a \to b$.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

2. Misalkan

$$s \equiv [d/c], \text{ dan}$$
$$\tau \equiv (a \to c) \to (b \to c),$$
$$s(\tau) \equiv (a \to d) \to (b \to d).$$

5.1.2 Notasi Penulisan Substitusi Tipe

Suatu substitusi tipe dinyatakan dengan huruf "r", "s", "t", "u", "v". Misalkan $s \equiv [\sigma_1/a_1, \ldots, \sigma_n/a_n]$, notasi penulisan lain untuk $s(\tau)$ adalah sebagai berikut:

$$[\sigma_1/a_1, \ldots, \sigma_n/a_n]\tau.$$

Kemudian, notasi $s \equiv [\sigma_1/a_1, \ldots, \sigma_n/a_n]$ dibaca "secara bersamaan menggantikan a_1 dengan σ_1, ..., a_n dengan σ_n".

5.1.3 Istilah dan Definisi Berkaitan dengan Konsep Substitusi Tipe

Sub-bab ini memaparkan beberapa istilah dan definisi yang cukup penting berkaitan dengan konsep substitusi tipe.

5.1.3.1 Komponen Substitusi

Untuk suatu substitusi

$$s \equiv [\sigma_1/a_1, \ldots, \sigma_n/a_n],$$

setiap ekspresi σ_i/a_i disebut sebagai ***komponen*** dari substitusi s. σ_i/a_i disebut *Trivial* jika $\sigma_i \equiv a_i$.

5.1.3.2 Domain dan Range Dalam Substitusi Tipe

Untuk suatu substitusi

$$s \equiv [\sigma_1/a_1, \ldots, \sigma_n/a_n],$$

himpunan $\{a_1, \ldots, a_n\}$ disebut sebagai ***variable-domain*** dari s, atau $Dom(s)$, dan $Vars(\sigma_1, \ldots, \sigma_n)$ disebut sebagai ***variable-range*** dari s, atau $Range(s)$.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

5.1.3.3 Substitusi Tunggal dan Kosong

Suatu substitusi

$$\boldsymbol{s} \equiv [\boldsymbol{\sigma_1/a_1, \ldots, \sigma_n/a_n}],$$

disebut:

- **Substitusi Kosong (*Empty Substitution*)** $\boldsymbol{e}$, jika n = 0. Sehingga $\boldsymbol{e(\tau)} \equiv \boldsymbol{\tau}$.
- **Substitusi Tunggal (*Single Substitution*)**, jika n = 1.

5.1.3.4 Substitusi s yang Terbatas pada V ($s \upharpoonright V$)

Jika substitusi $\boldsymbol{s} \equiv [\boldsymbol{\sigma_1/a_1, \ldots, \sigma_n/a_n}]$ dan $\boldsymbol{V}$ adalah himpunan variabel, maka $\boldsymbol{s} \upharpoonright \boldsymbol{V}$ memiliki arti bahwa $\boldsymbol{s}$ merupakan operator substitusi yang memiliki komponen $\boldsymbol{\sigma_i/a_i}$ sedemikian sehingga $\boldsymbol{a_i} \in \boldsymbol{V}$.

Lemma: $(\boldsymbol{s} \upharpoonright \boldsymbol{Vars(\tau)})(\boldsymbol{\tau}) \equiv \boldsymbol{s(\tau)}$

5.1.3.5 Substitusi *Variable* dengan *Variable*

Substitusi *variable* dengan *variable* merupakan substitusi $\boldsymbol{s} \equiv [\boldsymbol{b_1/a_1, \ldots, b_n/a_n}]$ dimana $\boldsymbol{b_1, \ldots, b_n}$ adalah *variable* (tidak harus berbeda). Jika $\boldsymbol{b_1, \ldots, b_n}$ berbeda, maka $\boldsymbol{s}$ disebut ***one-to-one***. Jika $\boldsymbol{b_1, \ldots, b_n}$ berbeda dan $\{\boldsymbol{a_1, \ldots, a_n}\} = \boldsymbol{Vars(\tau)}$ untuk suatu $\boldsymbol{\tau}$, maka $\boldsymbol{s}$ disebut operasi ***renaming di*** $\boldsymbol{\tau}$.

Definisi yang lainnya adalah, misalkan $\boldsymbol{\sigma}$ adalah sebuah tipe, $\boldsymbol{\sigma}$ disebut ***alphabetic variant*** dari $\boldsymbol{\tau}$, atau $\boldsymbol{\sigma}$ dan $\boldsymbol{\tau}$ disebut ***identical modulo renaming*** jika dan hanya jika $\boldsymbol{\sigma} \equiv \boldsymbol{s(\tau)}$ untuk suatu operator *renaming* $\boldsymbol{s}$ yang diterapkan pada $\boldsymbol{\tau}$. Kemudian misalkan $\boldsymbol{\tau}$ adalah *principal-type* dari *term* $\boldsymbol{M}$, $\boldsymbol{\tau}$ disebut ***unique modulo renaming***. Dalam artian bahwa tipe lain $\boldsymbol{\sigma}$ adalah *principal-type* dari $\boldsymbol{M}$ jika dan hanya jika $\boldsymbol{\sigma}$ adalah *alphabetic variant* dari $\boldsymbol{\tau}$.

5.1.4 Substitusi Tipe Pada Barisan Tipe, *Context*, Formula TA$_\lambda$, dan deduksi (∇)

Selain substitusi pada tipe, terdapat juga substitusi untuk barisan tipe, *context*, formula TA$_\lambda$, dan deduksi(∇). Berikut ini adalah definisi substitusi tersebut:

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

- Substitusi pada barisan tipe

$$s(\langle\tau_1, \ldots, \tau_n\rangle) \;=\; \langle s(\tau_1), \ldots, s(\tau_n)\rangle.$$

- Substitusi pada *type-context*

$$s(\Gamma) \;=\; \{x_1 : s(\tau_1), \ldots, x_m : s(\tau_m)\}, \qquad \text{Jika } x_1 : \tau_1, \ldots, x_m : \tau_m,$$

  (konsistensi pada Γ tetap terjaga pada $s(\Gamma)$.

- Substitusi pada formula TA$_\lambda$

$$s(\Gamma \mapsto M : \tau) \;=\; s(\Gamma) \;\mapsto\; M : s(\tau),$$

- Substitusi pada deduksi ∇, $s(\nabla)$, adalah hasil menerapkan s pada setiap formula TA$_\lambda$ dalam ∇. Kita sebut $s(\nabla)$ sebagai *instance* dari ∇.

  (jika ∇ adalah deduksi TA$_\lambda$ maka $s(\nabla)$ juga merupakan deduksi TA$_\lambda$.

5.1.5 Gabungan (*Union*) Substitusi Tipe

Jika

$$s \;\equiv\; [\sigma_1/a_1, \ldots, \sigma_n/a_n] \text{ dan } t \;\equiv\; [\tau_1/b_1, \ldots, \tau_p/b_p],$$

serta $a_1, \ldots, a_n, b_1, \ldots, b_p$ adalah *type-variabel* yang berbeda, atau berlaku $a_i \;\equiv\; b_j \Rightarrow \sigma_i \;\equiv\; \tau_j$, maka kita definisikan

$$s \,\cup\, t \;\equiv\; [\sigma_1/a_1, \ldots, \sigma_n/a_n, \tau_1/b_1, \ldots, \tau_p/b_p]$$

(dimana pengulangan dihilangkan).

5.1.6 Komposisi Substitusi Tipe

Komposisi dari dua substitusi s dan t adalah substitusi yang dilakukan secara bersamaan, yang memiliki efek sama persis dengan menerapkan substitusi t terlebih dahulu dan kemudian menerapkan substitusi s. Secara lebih *formal* komposisi dari dua substitusi didefinisikan sebagai berikut:

Misalkan s dan t adalah suatu substitusi,

$s \;\equiv\; [\sigma_1/a_1, \ldots, \sigma_n/a_n]$, dan

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

$$t \equiv [\tau_1/b_1, \ldots, \tau_p/b_p]$$

kemudian kita definisikan

$$s \circ t \equiv [\sigma_{i1}/a_{i1}, \ldots, \sigma_{ih}/a_{ih}, s(\tau_1)/b_1, \ldots, s(\tau_p)/b_p]$$

dimana $\{a_{i1}, \ldots, a_{ih}\} = Dom(s) - Dom(t)$ dan $0 \leq h \leq n$.

5.2 Definisi *Principal Type* (PT)

Secara umum sebuah *typable term* memiliki sejumlah tak hingga tipe dalam TA_λ. Misalnya untuk $I = \lambda x.x$, kita dapat memberikan setiap tipe yang memiliki bentuk $\sigma \to \sigma$ melalui deduksi seperti berikut ini

$$\frac{x : \sigma \mapsto x : \sigma}{\mapsto (\lambda x.x) : \sigma \to \sigma} (\to I)_{main}$$

Semua tipe yang berjumlah tak berhingga ini adalah *instance* hasil substitusi pada satu tipe a→a. Kemudian berdasarkan teorema konstruksi subjek, setiap deduksi untuk I pastilah memiliki bentuk seperti yang ditunjukkan di atas. Sehingga bisa dikatakan bahwa I tidak memiliki tipe lain selain a→a, dan tipe tersebut disebut *principal-type* dari I. *Principal-type* bisa dikatakan sebagai tipe yang paling umum yang bisa diterima suatu *term* dalam TA_λ, dan ini merupakan properti dari semua *typable-term.*

Berikut ini adalah definisi yang lebih formal dari *principal-type.* Dalam TA_λ, ***Principal Type*** (PT) dari sebuah *term* M adalah tipe τ sedemikian sehingga:

1. $\Gamma \vdash_\lambda M : \tau$ untuk suatu Γ.

2. Jika $\Gamma' \vdash_\lambda M : \sigma$ untuk suatu Γ' dan σ, maka σ adalah *instance* dari τ.

   Sebuah tipe τ adalah PT dari M, jika dan hanya jika untuk semua tipe σ berlaku

   $$(\exists \Gamma')(\Gamma' \vdash_\lambda M : \sigma) \iff \sigma \text{ adalah } instance \text{ dari } \tau$$

   jadi, dengan kata lain bisa dikatakan bahwa PT dari M merupakan tipe yang mewakili himpunan semua tipe yang bisa diberikan pada M.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

Principal type dari suatu *term* adalah unik. Kemudian dalam penulisannya, untuk suatu $\boldsymbol{\lambda}$-*term* $\boldsymbol{M}$, "*Principal type* dari $\boldsymbol{M}$" dituliskan sebagai berikut

$$\boldsymbol{PT(M)}$$

5.3 Definisi *Principal Pair*

Principal pair dari *term* $\boldsymbol{M}$ adalah sebuah pasangan $\langle \boldsymbol{\Gamma}, \boldsymbol{\tau} \rangle$ sedemikian sehingga formula $\boldsymbol{\Gamma} \mapsto \boldsymbol{M} : \boldsymbol{\tau}$ dapat dideduksi dalam $\mathrm{TA}_{\boldsymbol{\lambda}}$, dan setiap formula lain $\boldsymbol{\Gamma} \mapsto \boldsymbol{M} : \boldsymbol{\sigma}$ yang dapat dideduksi dalam $\mathrm{TA}_{\boldsymbol{\lambda}}$ adalah *instance* dari $\boldsymbol{\Gamma} \mapsto \boldsymbol{M} : \boldsymbol{\tau}$.

5.4 Definisi *Principal Deduction*

Principal deduction dari sebuah *term* $\boldsymbol{M}$ adalah sebuah deduksi $\boldsymbol{\nabla}$ dari suatu formula $\boldsymbol{\Gamma} \mapsto \boldsymbol{M} : \boldsymbol{\tau}$, sedemikian sehingga setiap deduksi lain yang konklusinya memiliki subjek $\boldsymbol{M}$ adalah *instance* dari $\boldsymbol{\nabla}$. Untuk lebih singkat dalam menyatakan "Ada sebuah *principal deduction* dari $\boldsymbol{\Gamma} \mapsto \boldsymbol{M} : \boldsymbol{\tau}$", kita tuliskan sebagai berikut:

$$\boldsymbol{\Gamma} \vdash_{\boldsymbol{p}} \boldsymbol{M} : \boldsymbol{\tau}.$$

Kemudian jika $\boldsymbol{\nabla}$ adalah *principal deduction* dari $\boldsymbol{\Gamma} \mapsto \boldsymbol{M} : \boldsymbol{\tau}$, maka $\boldsymbol{\tau}$ adalah *principal type* dari $\boldsymbol{M}$ dan $\langle \boldsymbol{\Gamma}, \boldsymbol{\tau} \rangle$ adalah *principal pair* untuk $\boldsymbol{M}$.

5.5 *Common Instance* dan *Most General Common Instance* (MGCI)

Sub-bab ini menjelaskan definisi serta pengertian dari *common instance* dan *most general common instance.* Dua konsep ini merupakan konsep yang penting dalam memahami algoritma PT.

Adapun definisi ***common instance*** adalah sebagai berikut:

1. $\boldsymbol{v}$ adalah ***common instance***$\boldsymbol{(c.i)}$ dari pasangan tipe $\langle \boldsymbol{\rho}, \boldsymbol{\tau} \rangle$, jika dan hanya jika

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

$$v \equiv s_1(\rho) \equiv s_2(\tau).$$

Kemudian kita sebut $\langle s_1, s_2 \rangle$ sebagai sepasang ***converging substitutions*** untuk $\langle \rho, \tau \rangle$.

2. $\langle v_1, \ldots, v_n \rangle$ adalah ***common instance***$(c.i)$ dari $\langle \rho_1, \ldots, \rho_n \rangle$ dan $\langle \tau_1, \ldots, \tau_n \rangle$, jika dan hanya jika

$$\langle v_1, \ldots, v_n \rangle \equiv s_1(\langle \rho_1, \ldots, \rho_n \rangle) \equiv s_2(\langle \tau_1, \ldots, \tau_n \rangle).$$

Kemudian kita sebut $\langle s_1, s_2 \rangle$ sebagai sepasang ***converging substitutions*** untuk $\langle \rho, \tau \rangle$.

Contoh: *Common instance* dari pasangan $\langle a \rightarrow (b \rightarrow c), (a \rightarrow b) \rightarrow a \rangle$ adalah tipe $((\beta \rightarrow \gamma) \rightarrow \delta) \rightarrow (\beta \rightarrow \gamma)$ (dimana β, γ, δ adalah sembarang tipe), dan *converging substitution*-nya adalah

$$s_1 \equiv [((\beta \rightarrow \gamma) \rightarrow \delta)/a,\ \beta/b,\ \gamma/c] \text{ dan } s_2 \equiv [(\beta \rightarrow \gamma)/a,\ \delta/b].$$

Selanjutnya, definisi ***most general common instance***$(MGCI)$ adalah sebagai berikut: *most general common instance* dari $\langle \rho, \tau \rangle$ adalah *common instance* v_0 sedemikian sehingga setiap *common instance* adalah *instance* dari v_0. Jika v_0 adalah sebuah MGCI dari $\langle \rho, \tau \rangle$ maka kita sebut pasangan $\langle s_1, s_2 \rangle$ (yang menyebabkan $s_1(\rho) \equiv s_2(\tau) \equiv v_0$) sebagai ***MGCI-generator*** dari $\langle \rho, \tau \rangle$.

5.6 Unifikasi

Sub-bab ini menjelaskan konsep unifikasi, salah satu konsep yang penting dalam algoritma PT. Penjelasan pada sub-bab ini juga disertai dengan pemaparan algoritma unifikasi yang digunakan dalam algoritma PT.

Secara garis besar, unifikasi dari sepasang tipe $\langle \rho, \tau \rangle$ mirip dengan *common instance*, hanya saja unifikasi didapat dari proses substitusi dengan menerapkan substitusi yang sama pada ρ dan τ.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

5.6.1 *Unifier*

Berikut ini adalah definisi dari ***unifier***:

- $\langle \rho, \tau \rangle$ disebut ***unifiable*** jika dan hanya jika terdapat substitusi s sedemikian sehingga berlaku $s(\rho) \equiv s(\tau)$, kemudian kita sebut s sebagai ***unifier*** dari $\langle \rho, \tau \rangle$ dan $s(\rho)$ sebagai ***unifikasi*** dari $\langle \rho, \tau \rangle$.

- ***Unifier*** dari sepasang barisan tipe $\langle \langle \rho_1, \ldots, \rho_n \rangle, \langle \tau_1, \ldots, \tau_n \rangle \rangle$ yang memiliki panjang yang sama adalah s, sedemikian sehingga:

$$s(\langle \rho_1, \ldots, \rho_n \rangle) \equiv s(\langle \tau_1, \ldots, \tau_n \rangle) .$$

Permasalahan menemukan *unifier* untuk pasangan barisan tipe dapat direduksi menjadi permasalahan menemukan *unifier* untuk sepasang tipe sebagai berikut: Misalkan diberikan dua barisan $\langle \rho_1, \ldots, \rho_n \rangle$ dan $\langle \tau_1, \ldots, \tau_n \rangle$, pilih sembarang *variable* b yang tidak berada kedua barisan tipe tersebut, kemudian definisikan

$$\rho^* \equiv \rho_1 \to \ldots \to \rho_n \to b \quad \tau^* \equiv \tau_1 \to \ldots \to \tau_n \to b.$$

Kemudian *unifier* kedua barisan tersebut adalah substitusi s jika dan hanya jika $\langle \rho^*, \tau^* \rangle$ ter-*unify* oleh $s \restriction Vars(\rho_1, \ldots, \rho_n, \tau_1, \ldots, \tau_n)$.

5.6.2 *Most General Unifier* dan *Most General Unification*

Most general unifier dari sepasang tipe $\langle \rho, \tau \rangle$ adalah *unifier* u sedemikian sehingga untuk setiap *unifier* s dari $\langle \rho, \tau \rangle$ yang lain berlaku

$$s(\rho) \equiv s'(u(\rho))$$

untuk suatu s'. Jika $v \equiv u(\rho)$ untuk suatu *most general unifier* u dari $\langle \rho, \tau \rangle$, maka kita sebut v sebagai ***most general unification*** dari $\langle \rho, \tau \rangle$.

5.6.3 Algoritma Unifikasi

Menurut teorema unifikasi (J. A. Robinson) ,

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

1. Terdapat algoritma yang dapat menentukan apakah pasangan $\langle \boldsymbol{\rho}, \boldsymbol{\tau} \rangle$ memiliki *unifier*, dan jika jawabannya adalah 'ya', maka algoritma tersebut akan membuat *most general unifier*-nya.

2. Jika sebuah pasangan $\langle \boldsymbol{\rho}, \boldsymbol{\tau} \rangle$ memiliki *unifier*, maka pasangan tersebut juga memiliki *most general unifier*.

3. Bagian (1)-(2) berlaku juga untuk pasangan deduksi dan pasangan barisan tipe yang berhingga.

Sub-bab ini menjelaskan algoritma unifikasi Robinson (1965). Berikut adalah penjelasannya:

Secara garis besar, yang dilakukan algoritma tersebut adalah mencoba membuat $\boldsymbol{u}$ (*Most General Unifier*) secara bertahap mulai dari $\boldsymbol{u_0}, \boldsymbol{u_1}, \ldots$ dst. Setiap $\boldsymbol{u_k}$ merupakan komposisi dari $\boldsymbol{u_{k-1}}$ dengan tambahan operator substitusi yang baru. Pada setiap langkah ke k, dilakukan pengecekan apakah $\boldsymbol{u_k}(\boldsymbol{\rho}) \equiv \boldsymbol{u_k}(\boldsymbol{\tau})$. Jika jawabannya adalah 'ya', maka $\boldsymbol{u} \equiv \boldsymbol{u_k}$ dan algoritma berhenti. Namun jika jawabannya adalah 'tidak', maka $\boldsymbol{u_k}$ dikembangkan lagi dan dilanjutkan pada tahap berikutnya.

Adapun algoritma unifikasi tersebut adalah sebagai berikut [Hin97] [Rob65]:

Algoritma Unifikasi Robinson (1965)

Input. Pasangan tipe $\langle \boldsymbol{\rho}, \boldsymbol{\tau} \rangle$.

***Output* yang diharapkan.**

- Kalimat yang absah (benar) bahwa $\langle \boldsymbol{\rho}, \boldsymbol{\tau} \rangle$ tidak dapat di *unify*, atau
- *Most General Unifier* $\boldsymbol{u}$ dari $\langle \boldsymbol{\rho}, \boldsymbol{\tau} \rangle$.

Langkah 0. Pilih $\boldsymbol{k} = \boldsymbol{0}$ dan $\boldsymbol{u_0} \equiv \boldsymbol{e}$ (substitusi kosong atau *empty substitution*).

Langkah k+1.

1. Diberikan $\boldsymbol{k}$ dan $\boldsymbol{u_k}$,

2. Buat $\boldsymbol{\rho_k} \equiv \boldsymbol{u_k}(\boldsymbol{\rho})$ dan $\boldsymbol{\tau_k} \equiv \boldsymbol{u_k}(\boldsymbol{\tau})$.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

3. Terapkan ***comparison procedure*** (*Comparison Procedure* dijelaskan pada sub-bab 5.6.3.1) pada $\langle \boldsymbol{\rho_k}, \boldsymbol{\tau_k} \rangle$. ***Comparison procedure*** akan menghasilkan *output*:

   - Kalimat yang absah (benar) bahwa $\boldsymbol{\rho_k \equiv \tau_k}$,

     Atau,

   - Sebuah ***disagreement pair*** $\langle \boldsymbol{a}, \boldsymbol{\alpha} \rangle$ sedemikian sehingga $\boldsymbol{\alpha \not\equiv a}$.

4. Dari hasil *comparison procedure* tersebut, langkah berikutnya adalah:

   - Jika $\boldsymbol{\rho_k \equiv \tau_k}$, maka pilih $\boldsymbol{u \equiv u_k}$, dan algoritma berhenti.
   - Jika $\boldsymbol{\rho_k \not\equiv \tau_k}$, maka *comparison procedure* pasti menghasilkan $\langle \boldsymbol{a}, \boldsymbol{\alpha} \rangle$. Selanjutnya cek apakah $\boldsymbol{a \in Vars(\alpha)}$.
     - Jika $\boldsymbol{a \in Vars(\alpha)}$, maka $\langle \boldsymbol{\rho}, \boldsymbol{\tau} \rangle$ tidak dapat di-*unify*, dan algoritma berhenti
     - Jika $\boldsymbol{a \notin Vars(\alpha)}$, maka:

       (a) Gantikan $\boldsymbol{k}$ dengan $\boldsymbol{k+1}$,

       (b) kemudian pilih $\boldsymbol{u_{k+1} \equiv [\alpha/a] \circ u_k}$,

       (c) dan lanjutkan ke langkah $\boldsymbol{k+2}$.

5.6.3.1 *Comparison Procedure*

Sub-bab ini menjelaskan ***comparison procedure*** yang merupakan bagian dari algoritma unifikasi Robinson.

Comparison Procedure [Hin97]

1. Diberikan sepasang tipe $\langle \boldsymbol{\mu}, \boldsymbol{v} \rangle$ (*Comparison Procedure* ini mensyaratkan bahwa tipe yang dimasukkan sebagai input harus dalam bentuk tanpa penyingkatan(tidak ada penghilangan tanda kurung)),

2. Tuliskan $\boldsymbol{\mu}$ dan $\boldsymbol{v}$ sebagai barisan karakter simbol (*symbol-string*), misalkan:

$$\boldsymbol{\mu \equiv s_1 \ldots s_m, \quad v \equiv t_1 \ldots t_n \quad (m, n \geq 1)}$$

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

dimana setiap $\boldsymbol{s_1 \ldots s_m, t_1 \ldots t_n}$ adalah kemunculan dari tanda kurung,tanda panah, dan *type-variable*.

3. Kemudian langkah selanjutnya adalah:

   - Jika $\boldsymbol{\mu \equiv v}$, maka nyatakan (keluarkan) output $\boldsymbol{\mu \equiv v}$, dan algoritma berhenti.
   - Jika $\boldsymbol{\mu \not\equiv v}$, maka:

     (a) Pilih $\boldsymbol{p}$ dengan nilai terkecil, dimana $\boldsymbol{p \leq Min\{m, n\}}$, sedemikian sehingga $\boldsymbol{s_p \not\equiv t_p}$ (satu dari $\boldsymbol{s_p}$ atau $\boldsymbol{t_p}$ pastilah merupakan *variable*, dan yang lainnya adalah "kurung buka '(' " atau *variable yang lain*). Kemudian $\boldsymbol{s_p}$ pastilah merupakan simbol paling kiri dari subtipe $\boldsymbol{\mu^*}$ yang unik dari $\boldsymbol{\mu}$ (Jika $\boldsymbol{s_p}$ adalah *variable*, maka $\boldsymbol{\mu* \equiv s_p}$). Sama seperti $\boldsymbol{s_p}$, $\boldsymbol{t_p}$ pastilah merupakan simbol paling kiri dari subtipe $\boldsymbol{v^*}$ yang unik dari $\boldsymbol{v}$.

     (b) Pilih satu dari $\boldsymbol{\mu^*}$ atau $\boldsymbol{v^*}$ yang merupakan *type-variable*, dan sebut sebagai "$\boldsymbol{a}$", kemudian sebut sisa dari $\boldsymbol{\langle \mu^*, v^* \rangle}$ sebagai "$\boldsymbol{\alpha}$". Pasangan $\boldsymbol{\langle a, \alpha \rangle}$ disebut ***disagreement pair*** untuk $\boldsymbol{\langle \mu, v \rangle}$

5.6.3.2 Contoh Penerapan Algoritma Unifikasi

Berikut adalah contoh penerapan algoritma unifikasi pada $\boldsymbol{\langle (a \to (b \to b)), ((c \to c) \to a) \rangle}$:

Langkah 0

$\boldsymbol{u_0 = e}$ (*empty substitution*),

$\boldsymbol{\rho_0 = (a \to (b \to b))}$,

$\boldsymbol{\tau_0 = ((c \to c) \to a)}$.

Langkah 1 Mulai konstruksi $\boldsymbol{\rho_1}$ dan $\boldsymbol{\tau_1}$. Didapatkan:

$\boldsymbol{\rho_1 \equiv u_0(\rho_0) \equiv \rho_0}$, dan

$\boldsymbol{\tau_1 \equiv u_0(\tau_0) \equiv \tau_0}$.

Terapkan *comparison procedure* pada $\boldsymbol{\rho_1}$ dan $\boldsymbol{\tau_1}$. Diperoleh:

$\boldsymbol{\mu^* \equiv a}$, $\boldsymbol{\nu^* \equiv (c \to c)}$ dan

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

$\langle \boldsymbol{a}, \boldsymbol{\alpha} \rangle \equiv \langle \boldsymbol{a}, (\boldsymbol{c} \to \boldsymbol{c}) \rangle$.

Karena $\boldsymbol{a} \notin \boldsymbol{Vars}(\boldsymbol{\alpha})$, mulai konstruksi $\boldsymbol{u_1}$. Didapatkan $\boldsymbol{u_1} \equiv [\boldsymbol{\alpha}/\boldsymbol{a}] \circ \boldsymbol{u_0} \equiv [(\boldsymbol{c} \to \boldsymbol{c})/\boldsymbol{a}]$. Kemudian lanjutkan ke langkah 2.

Langkah 2 Mulai konstruksi $\boldsymbol{\rho_2}$ dan $\boldsymbol{\tau_2}$. Didapatkan:

$\boldsymbol{\rho_2} \equiv \boldsymbol{u_1}(\boldsymbol{\rho_1}) \equiv ((\boldsymbol{c} \to \boldsymbol{c}) \to (\boldsymbol{b} \to \boldsymbol{b}))$, dan

$\boldsymbol{\tau_2} \equiv \boldsymbol{u_1}(\boldsymbol{\tau_1}) \equiv ((\boldsymbol{c} \to \boldsymbol{c}) \to (\boldsymbol{c} \to \boldsymbol{c}))$.

Terapkan *comparison procedure* pada $\boldsymbol{\rho_2}$ dan $\boldsymbol{\tau_2}$. Diperoleh:

$\boldsymbol{\mu}^* \equiv \boldsymbol{b}$, $\boldsymbol{\nu}^* \equiv \boldsymbol{c}$ dan

$\langle \boldsymbol{a}, \boldsymbol{\alpha} \rangle \equiv \langle \boldsymbol{b}, \boldsymbol{c} \rangle$.

Karena $\boldsymbol{a} \notin \boldsymbol{Vars}(\boldsymbol{\alpha})$, mulai konstruksi $\boldsymbol{u_2}$. Didapatkan $\boldsymbol{u_2} \equiv [\boldsymbol{\alpha}/\boldsymbol{a}] \circ \boldsymbol{u_1} \equiv [\boldsymbol{c}/\boldsymbol{b}] \circ [(\boldsymbol{c} \to \boldsymbol{c})/\boldsymbol{a}] \equiv [\boldsymbol{c}/\boldsymbol{b}, (\boldsymbol{c} \to \boldsymbol{c})/\boldsymbol{a}]$. Kemudian lanjutkan ke langkah 3.

Langkah 3 Mulai konstruksi $\boldsymbol{\rho_3}$ dan $\boldsymbol{\tau_3}$. Didapatkan:

$\boldsymbol{\rho_3} \equiv \boldsymbol{u_2}(\boldsymbol{\rho_2}) \equiv ((\boldsymbol{c} \to \boldsymbol{c}) \to (\boldsymbol{c} \to \boldsymbol{c}))$, dan

$\boldsymbol{\tau_3} \equiv \boldsymbol{u_2}(\boldsymbol{\tau_2}) \equiv ((\boldsymbol{c} \to \boldsymbol{c}) \to (\boldsymbol{c} \to \boldsymbol{c}))$.

Terapkan *comparison procedure* pada $\boldsymbol{\rho_3}$ dan $\boldsymbol{\tau_3}$. Diperoleh:

$\boldsymbol{\rho_3} \equiv \boldsymbol{\tau_3}$

Oleh karena itu didapatkan $\boldsymbol{u} \equiv \boldsymbol{u_2} \equiv [\boldsymbol{c}/\boldsymbol{b}, (\boldsymbol{c} \to \boldsymbol{c})/\boldsymbol{a}]$.

5.7 Ide (Prinsip) Dasar untuk Bagian Inti dari Algoritma *Principal Type* (PT)

Dua langkah utama dari algoritma PT adalah komputasi $\boldsymbol{PT}(\boldsymbol{\lambda x.P})$ dari $\boldsymbol{PT}(\boldsymbol{P})$ dan komputasi $\boldsymbol{PT}(\boldsymbol{PQ})$ dari $\boldsymbol{PT}(\boldsymbol{P})$ dan $\boldsymbol{PT}(\boldsymbol{Q})$. Komputasi $\boldsymbol{PT}(\boldsymbol{PQ})$ dari $\boldsymbol{PT}(\boldsymbol{P})$ dan $\boldsymbol{PT}(\boldsymbol{Q})$ ini bisa dibilang sebagai inti dari algoritma PT. Bagian tersebut merupakan bagian yang tersulit, dan sub-bab ini bertujuan untuk menjelaskan ide dari bagian tersebut. Berikut adalah penjelasannya.

Misalkan kita mencoba menentukan apakah aplikasi $\boldsymbol{PQ}$ dapat menerima tipe atau tidak di dalam $\mathrm{TA}_{\boldsymbol{\lambda}}$, dan kita telah mengetahui bahwa

$$\boldsymbol{PT}(\boldsymbol{P}) \equiv \boldsymbol{\rho} \to \boldsymbol{\sigma}, \quad \boldsymbol{PT}(\boldsymbol{Q}) \equiv \boldsymbol{\tau}.$$

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

Tipe yang dapat diberikan pada P dihasilkan dari $\rho \rightarrow \sigma$ melalui suatu substitusi, dan tipe yang dapat diberikan pada Q dihasilkan dari τ melalui suatu substitusi. Kemudian, jika kita dapat menemukan substitusi s_1 dan s_2 sedemikian sehingga

$$s_1(\rho) \equiv s_2(\tau),$$

maka kita dapat menerapkan aturan $(\rightarrow)$ untuk melakukan deduksi PQ seperti berikut ini:

$$\frac{\mapsto P : s_1(\rho) \rightarrow s_1(\sigma) \qquad \mapsto Q : s_2(\tau)}{\mapsto PQ : s_1(\sigma)} (\rightarrow E)$$

Jadi, masalah untuk menentukan apakah PQ dapat diberikan tipe atau tidak direduksi menjadi masalah menemukan substitusi s_1 dan s_2 sedemikian sehingga $s_1(\rho) \equiv s_2(\tau)$, atau dengan kata lain berdasarkan pengertian *common instance* yang dijelaskan pada sub-bab 5.5, masalah menemukan $PT(PQ)$ direduksi menjadi menemukan MGCI dari $\langle \rho, \tau \rangle$. Sebut saja

$$v \equiv s_1(\rho) \equiv s_2(\tau)$$

dan kemudian $PT(PQ) \equiv s_1(\sigma)$.

Namun, masih terdapat permasalahan dalam hal ini. Misalkan σ mengandung *variable* $b_1, \ldots, b_k$ yang tidak muncul pada ρ, tapi ternyata MGCI v juga mengandung satu atau beberapa *variable* tersebut. Hal tersebut menyebabkan $s_1(\sigma)$ dapat memiliki 2 kemunculan *variable* b_i, satu *variable* asli dari σ, dan yang lain adalah dari s_1. Dalam kasus ini $s_1(\sigma)$ tidak akan menjadi tipe paling umum yang bisa diberikan kepada PQ. Hal ini dikarenakan kita dapat mengganti v menjadi variasi *alphabetic* $v' \equiv s_1'(\rho)$ tanpa *variable* yang sama dengan yang ada pada σ, dan $s_1'(\sigma)$ menjadi tipe dari PQ yang bukan merupakan *instance* dari $s_1(\sigma)$.

Contoh konkritnya adalah sebagai berikut:

Misalkan $PT(P) = a \rightarrow (b \rightarrow a)$ dan $PT(Q) = b \rightarrow b$.

$$\rho = a, \ \sigma = b \rightarrow a, \ \tau = b \rightarrow b.$$

MGCI dari $\langle \rho, \tau \rangle$ adalah $v \equiv b \rightarrow b$, didapatkan dari ρ dengan substitusi $s_1 = [(b \rightarrow b)/a]$. Oleh karena itu $s_1(\sigma) = s_1(b \rightarrow a) = b \rightarrow (b \rightarrow b)$. Tapi $b \rightarrow (b \rightarrow b)$ bukanlah *principal type* dari PQ. Karena jika kita mengubah s_1

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

menjadi sebuah substitusi yang baru s_1' dengan menggantikan b dengan *variable* baru c yang tidak ada di σ, maka kita dapatkan

$$\vdash_\lambda P : (c \to c) \to (b \to (c \to c)) \qquad \vdash_\lambda Q : c \to c$$

dan PQ memiliki tipe $b \to (c \to c)$ yang bukan merupakan *instance* dari $b \to (b \to b)$.

Untuk mengatasi permasalahan tersebut, algoritma PT akan mencari MGCI v dari $\langle \rho, \tau \rangle$ dengan hati-hati sedemikian sehingga

$$Vars(v) \cap (Vars(\sigma) - Vars(\rho)) = \emptyset.$$

Seperti yang dijelaskan, sekarang permasalahan menemukan $PT(PQ)$ direduksi menjadi permasalahan mencari MGCI dari $\langle \rho, \tau \rangle$. Untuk itu, kita membutuhkan suatu algoritma yang dapat menentukan apakah $\langle \rho, \tau \rangle$ memiliki *common instance.* Kemudian jika hasilnya "ya", maka algoritma tersebut akan menghasilkan c.i yang paling umum (MGCI). Algoritma PT akan menemukan MGCI ini secara tidak langsung dengan algoritma unifikasi Robinson.

Pada suatu kasus khusus saat ρ dan τ tidak memiliki *variable* yang sama, setiap *common instance* jugalah merupakan *unification*-nya. Oleh karena itu algoritma PT akan mencari *most general unification* dari $\langle \rho, \tau \rangle$ untuk mencari MGCI dari $\langle \rho, \tau \rangle$. Hal tersebut dikarenakan jika

$$v \equiv s_1(\rho) \equiv s_2(\tau)$$

dan $Dom(s_1) \subseteq Vars(\rho)$ serta $Dom(s_2) \subseteq Vars(\tau)$, maka $s_1 \cup s_2$ terdefinisikan dan

$$v \equiv s_1 \cup s_2(\rho) \equiv s_1 \cup s_2(\tau)$$

Fakta ini memberikan *lemma* sebagai berikut:

- Jika ρ dan τ tidak memiliki *variable* yang sama, $\langle \rho, \tau \rangle$ memiliki *most general unification* jika dan hanya jika $\langle \rho, \tau \rangle$ memiliki MGCI, dan keduanya identik.

- Untuk Semua ρ dan τ. Jika kita ubah τ ke variasi alfabetisnya τ^* dengan tidak ada *variable* yang sama di ρ, maka unifikasi dari $\langle \rho, \tau^* \rangle$ merupakan *common instance* dari $\langle \rho, \tau \rangle$, dan *most general unification* dari $\langle \rho, \tau^* \rangle$ akan menjadi MGCI dari $\langle \rho, \tau \rangle$.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

- Hal yang sama juga berlaku pada pasangan barisan tipe atau deduksi.

Oleh karena itu, sekarang permasalahan mencari MGCI telah direduksi menjadi mencari *most general unification* dari pasangan $\langle \boldsymbol{\rho}, \boldsymbol{\tau} \rangle$ yang keduanya tidak memiliki *variable* yang sama.

5.8 Penjelasan Algoritma *Principal Type* (PT)

Sub-bab ini menjelaskan algoritma *Principal Type* (PT). Algoritma ini berfungsi untuk menentukan apakah suatu *term* $\boldsymbol{M}$ dapat diberikan tipe (*typable*) dalam $\text{TA}_{\boldsymbol{\lambda}}$, dan jika jawabannya adalah 'ya', maka algoritma tersebut akan mengeluarkan *principal deduction* dan *principal type* untuk $\boldsymbol{M}$. Namun jika jawabannya 'tidak', maka algoritma ini akan mengeluarkan suatu pernyataan yang benar bahwa *term* tersebut tidak dapat diberikan tipe di $\text{TA}_{\boldsymbol{\lambda}}$. *Flow chart* algoritma PT ini berada pada Lampiran F, dan berikut ini adalah penjelasan algoritma tersebut [Hin97] [Hin69]:

Algoritma Principal Type (PT)

Input.

Sembarang $\boldsymbol{\lambda}$-*term* $\boldsymbol{M}$ (Baik *closed* ataupun tidak).

***Output* yang diharapkan.**

Principal dedcution $\boldsymbol{\nabla M}$ untuk $\boldsymbol{M}$ atau suatu pernyataan yang benar (absah) bahwa $\boldsymbol{M}$ tidak dapat diberikan tipe (*untypable*).

Kasus I. (Secara umum kasus I ini digambarkan pada Gambar F.2)

- Jika $\boldsymbol{M}$ adalah *term-variable*, misalnya $\boldsymbol{M} \equiv \boldsymbol{x}$,
- Maka pilih $\boldsymbol{\nabla}_{\boldsymbol{M}}$ sebagai satu formula deduksi $\boldsymbol{x} : \boldsymbol{a} \mapsto \boldsymbol{x} : \boldsymbol{a}$, dimana $\boldsymbol{a}$ adalah sembarang *type-variable* ($\boldsymbol{\nabla}_{\boldsymbol{M}}$ adalah *Principal Deduction* untuk $\boldsymbol{M}$).

Kasus II. (Secara umum kasus II ini digambarkan pada Gambar F.3)

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

- Jika $M \equiv \lambda x.P$ dan $x \in FV(P)$, misalkan $FV(P) = \{x, x_1, \ldots, x_t\}$,
- Terapkan algoritma ini pada P. Kemudian langkah selanjutnya adalah:
  - Jika P tidak dapat diberikan tipe, maka dapat disimpulkan bahwa M tidak dapat diberikan tipe juga.
  - Jika P memiliki *principal deduction* ∇_P, maka konklusi dari ∇_P tersebut pasti memiliki bentuk

    $$x : \alpha, x_1 : \alpha_1, \ldots, x_t : \alpha_t \mapsto P : \beta$$

    untuk suatu tipe $\alpha, \alpha_1, \ldots, \alpha_t, \beta$. Kemudian terapkan aturan $(\rightarrow I)_{main}$ untuk membuat deduksi:

    $$x_1 : \alpha_1, \ldots, x_t : \alpha_t \mapsto (\lambda x.P) : \alpha \rightarrow \beta$$

    kita sebut deduksi ini sebagai $\nabla_{\lambda xP}$

Kasus III. (Secara umum kasus III ini digambarkan pada Gambar F.4)

- Jika $M \equiv \lambda x.P$ dan $x \notin FV(P)$, misalkan $FV(P) = \{x_1, \ldots, x_t\}$,
- Terapkan algoritma ini pada P. Kemudian langkah selanjutnya adalah:
  - Jika P tidak dapat diberikan tipe (*not typable*), maka dapat disimpulkan bahwa M tidak dapat diberikan tipe juga (*not typable*).
  - Jika P memiliki *principal deduction* ∇_P, maka konklusi dari ∇_P tersebut pasti memiliki bentuk

    $$x_1 : \alpha_1, \ldots, x_t : \alpha_t \mapsto P : \beta$$

    untuk suatu tipe $\alpha_1, \ldots, \alpha_t, \beta$. Lalu pilih sebuah *type-variable* yang baru, misalkan d, yang tidak ada ∇_P. Kemudian terapkan aturan $(\rightarrow I)_{vac}$ untuk membuat deduksi:

    $$x_1 : \alpha_1, \ldots, x_t : \alpha_t \mapsto (\lambda x.P) : d \rightarrow \beta$$

    kita sebut deduksi ini sebagai $\nabla_{\lambda xP}$

Kasus IV. (Secara umum kasus IV ini digambarkan pada Gambar F.5)

- Jika $M \equiv PQ$, maka terapkan algoritma ini pada P dan Q, kemudian langkah selanjutnya adalah:

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

– Jika $\boldsymbol{P}$ atau $\boldsymbol{Q}$ tidak bisa diberikan tipe, maka bisa disimpulkan bahwa $\boldsymbol{M}$ tidak bisa diberikan tipe juga.

– Jika $\boldsymbol{P}$ dan $\boldsymbol{Q}$ keduanya dapat diberikan tipe (*typable*), misalkan mereka memiliki *principal deduction* $\boldsymbol{\nabla_P}$ dan $\boldsymbol{\nabla_Q}$, maka langkah selanjutnya adalah:

  1. Jika diperlukan, lakukan operasi *renaming type-variable* untuk memastikan bahwa $\boldsymbol{\nabla_P}$ dan $\boldsymbol{\nabla_Q}$ tidak memiliki *type-variable* yang sama.

  2. Selanjutnya, daftarkan *free-variable* di $\boldsymbol{P}$ dan di $\boldsymbol{Q}$ (daftar *free-variable* ini dapat saling *overlap*), misalkan

$$\boldsymbol{FV(P) = \{u_1, \ldots, u_p, w_1, \ldots, w_r\}} \qquad \boldsymbol{(p, r \geq 0)}$$
$$\boldsymbol{FV(Q) = \{v_1, \ldots, u_p, w_1, \ldots, w_r\}} \qquad \boldsymbol{(q \geq 0)}$$

  dimana $\boldsymbol{u_1, \ldots, u_p, v_1, \ldots, v_q, w_1, \ldots, w_r}$ adalah *type-variable* yang berbeda.

– Selanjutnya lanjutkan pada sub-kasus dari kasus IV ini (sub-kasus IVa atau IVb).

SubKasus IVa. (Secara umum kasus IVa ini digambarkan pada Gambar F.6, serta contoh penerapan kasus IVa digambarkan pada Gambar F.7).

1. Jika $\boldsymbol{M \equiv PQ}$ dan $\boldsymbol{PT(P)}$ adalah tipe yang *composite*, misalkan $\boldsymbol{PT(P) \equiv \rho \to \sigma}$. Maka konklusi dari $\boldsymbol{\nabla_P}$ dan $\boldsymbol{\nabla_Q}$ memiliki bentuk

$$\boldsymbol{u_1 : \theta_1, \ldots, u_p : \theta_p, w_1 : \psi_1, \ldots, w_r : \psi_r \mapsto P : \rho \to \sigma}$$
$$\boldsymbol{v_1 : \phi_1, \ldots, v_q : \phi_q, w_1 : \chi_1, \ldots, w_r : \chi_r \mapsto Q : \tau}$$

2. Terapkan algoritma unifikasi pada pasangan:

$$\boldsymbol{\langle \psi_1, \ldots, \psi_r, \rho \rangle}, \ \boldsymbol{\langle \chi_1, \ldots, \chi_r, \tau \rangle}$$

3. Selanjutnya lanjutkan pada sub-kasus dari kasus IVa (sub-kasus IVa1 atau IVa2).

SubKasus IVa1. Jika pasangan $\boldsymbol{\langle \psi_1, \ldots, \psi_r, \rho \rangle}$, $\boldsymbol{\langle \chi_1, \ldots, \chi_r, \tau \rangle}$ tidak memiliki *unifier*, maka $\boldsymbol{PQ}$ tidak bisa diberikan tipe.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

SubKasus IVa2. (Secara umum kasus IVa2 ini digambarkan pada Gambar F.8) Jika pasangan $\langle \psi_1, \ldots, \psi_r, \rho \rangle$, $\langle \chi_1, \ldots, \chi_r, \tau \rangle$ memiliki *unifier*. Maka algoritma unifikasi akan menghasilkan *most general unifier* u. Langkah selanjutnya adalah:

1. Lakukan *renaming* jika diperlukan untuk memastikan bahwa:

   $Dom(u) = Vars(\psi_1, \ldots, \psi_r, \rho, \chi_1, \ldots, \chi_r, \tau)$, dan

   $Range(u) \cap V = \emptyset$,

   dimana

   $$V = (Vars(\nabla_P) \cup Vars(\nabla_Q)) - Dom(u).$$

2. Terapkan u pada ∇_P dan ∇_Q, hal ini mengubah konklusi dari dua deduksi tersebut menjadi:

   $$u_1 : \theta_1^*, \ldots, u_p : \theta_p^*, w_1 : \psi_1^*, \ldots, w_r : \psi_r^* \ \mapsto \ P : \rho^* \to \sigma^*$$
   $$v_1 : \phi_1^*, \ldots, v_q : \phi_q^*, w_1 : \chi_1^*, \ldots, w_r : \chi_r^* \ \mapsto \ Q : \tau^*$$

   dimana $\theta_1^* \equiv u(\theta_1)$, dan seterusnya. Berdasarkan definisi u, kita memiliki :

   $$\psi_1^* \equiv \chi_1^*, \ldots, \psi_r^* \equiv \chi_r^*, \rho^* \equiv \tau^*$$

3. Selanjutnya, aturan ($\to$E) dapat diterapkan pada konklusi dari $u(\nabla_P)$ dan $u(\nabla_Q)$, kita sebut hasil kombinasi deduksi ini sebagai ∇_{PQ}, dan konklusinya adalah:

   $$u_1 : \theta_1^*, \ldots, u_p : \theta_p^*, v_1 : \phi_1^*, \ldots, v_q : \phi_q^*, w_1 : \chi_1^*, \ldots, w_r : \chi_r^* \ \mapsto \ PQ : \sigma^*$$

SubKasus IVb. (Secara umum kasus IVb ini digambarkan pada Gambar F.9, serta contoh penerapan kasus IVa digambarkan pada Gambar F.10)

1. Jika $M \equiv PQ$ dan $PT(P)$ adalah *atomic*, misalkan $PT(P) \equiv b$. Maka konklusi dari ∇_P dan ∇_Q memiliki bentuk

   $$u_1 : \theta_1, \ldots, u_p : \theta_p, w_1 : \psi_1, \ldots, w_r : \psi_r \ \mapsto \ P : b$$
   $$v_1 : \phi_1, \ldots, v_q : \phi_q, w_1 : \chi_1, \ldots, w_r : \chi_r \ \mapsto \ Q : \tau$$

2. Pilih sembarang *variable* $c \notin Vars(\nabla_P) \cup Vars(\nabla_Q)$, kemudian terapkan algoritma unifikasi pada pasangan:

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

$$\langle \psi_1, \ldots, \psi_r, b \rangle, \ \langle \chi_1, \ldots, \chi_r, \tau \rightarrow c \rangle$$

3. Selanjutnya lanjutkan pada sub-kasus dari kasus IVb (sub-kasus IVb1 atau IVb2).

SubKasus IVb1. Jika pasangan $\langle \psi_1, \ldots, \psi_r, b \rangle$, $\langle \chi_1, \ldots, \chi_r, \tau \rightarrow c \rangle$ tidak memiliki *unifier*, maka PQ tidak bisa diberikan tipe.

SubKasus IVb2. (Secara umum kasus IVb2 ini digambarkan pada Gambar F.11) Jika pasangan $\langle \psi_1, \ldots, \psi_r, b \rangle$, $\langle \chi_1, \ldots, \chi_r, \tau \rightarrow c \rangle$ memiliki *unifier.* Maka algoritma unifikasi akan menghasilkan *most general unifier* u. Langkah selanjutnya adalah:

1. Lakukan *renaming* jika diperlukan untuk memastikan bahwa:

   $Dom(u) = Vars(\psi_1, \ldots, \psi_r, b, \chi_1, \ldots, \chi_r, \tau \rightarrow c)$, dan

   $Range(u) \cap V = \emptyset$,

   dimana

   $V = (Vars(\nabla_P) \cup Vars(\nabla_Q)) - Dom(u)$.

2. Kemudian terapkan u pada ∇_P dan ∇_Q, berdasarkan definisi u, kita dapat

$$u(b) \ \equiv \ u(\tau \rightarrow c) \ \equiv \ u(\tau) \rightarrow u(c)$$

   dan konklusi dari dua deduksi tersebut menjadi:

$$u_1 : \theta_1^*, \ldots, u_p : \theta_p^*, w_1 : \psi_1^*, \ldots, w_r : \psi_r^* \ \mapsto \ P : \tau^* \rightarrow c^*$$
$$v_1 : \phi_1^*, \ldots, v_q : \phi_q^*, w_1 : \chi_1^*, \ldots, w_r : \chi_r^* \ \mapsto \ Q : \tau^*$$

   dimana * menandakan hasil penerapan dari u, berdasarkan definisi u, kita memiliki

$$\psi_1^* \ \equiv \ \chi_1^*, \ldots, \psi_r^* \ \equiv \ \chi_r^*$$

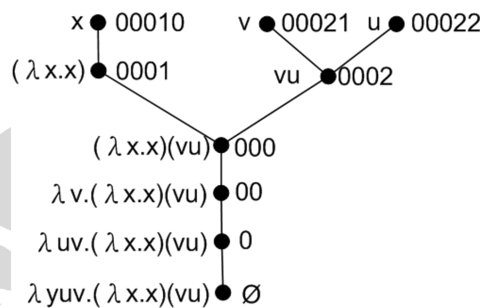
3. Selanjutnya, aturan ($\rightarrow$E) dapat diterapkan pada konklusi dari $u(\nabla_P)$ dan $u(\nabla_Q)$, kita sebut hasil kombinasi deduksi ini sebagai ∇_{PQ}, dan konklusinya adalah:

$$u_1 : \theta_1^*, \ldots, u_p : \theta_p^*, v_1 : \phi_1^*, \ldots, v_q : \phi_q^*, w_1 : \chi_1^*, \ldots, w_r : \chi_r^* \ \mapsto \ PQ : c^*$$

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

5.9 Contoh Penerapan Algoritma PT

Berikut adalah contoh penerapan algoritma PT pada $\boldsymbol{\lambda yuv.(\lambda x.x)(vu)}$. *Term* ini *typable* dan proses pemberian tipenya meng-*cover* semua kasus algoritma PT (I, II, III, IV, IVa, IVa2, IVb, IVb2), kecuali kasus ketika *term* yang diberikan *untypable.* Dalam penjelasan, referensi penomoran sub-*term* mengacu *parse tree term* yang disajikan pada Gambar 5.1.



Gambar 5.1: *Parse tree* untuk $\boldsymbol{\lambda yuv.(\lambda x.x)(vu)}$.

Berikut adalah tahapan penerapan algoritma PT untuk $\boldsymbol{M \equiv \lambda yuv.(\lambda x.x)(vu)}$:

1. Untuk $\boldsymbol{M \equiv \lambda yuv.(\lambda x.x)(vu)}$, terapkan kasus III, dengan $\boldsymbol{M \equiv \lambda yuv.(\lambda x.x)(vu)}$ dan $\boldsymbol{y \notin FV(P)}$, dimana $\boldsymbol{P \equiv \lambda uv.(\lambda x.x)(vu)}$ (0). Dalam kasus III, pertama kita harus menerapkan algoritma PT pada P, untuk mengecek apakah P *typable* atau tidak. Hal ini dijelaskan pada tahap 2.

2. Untuk *term* $\boldsymbol{(0) \equiv \lambda uv.(\lambda x.x)(vu)}$, terapkan kasus II, dengan $\boldsymbol{M \equiv \lambda uv.(\lambda x.x)(vu)}$ dan $\boldsymbol{u \in FV(P)}$, dimana $\boldsymbol{P \equiv \lambda v.(\lambda x.x)(vu)}$ (00). Dalam kasus II, pertama kita harus menerapkan algoritma PT pada P, untuk mengecek apakah P *typable* atau tidak. Hal ini dijelaskan pada tahap 3.

3. Untuk *term* $\boldsymbol{(00) \equiv \lambda v.(\lambda x.x)(vu)}$, terapkan kasus II, dengan $\boldsymbol{M \equiv \lambda v.(\lambda x.x)(vu)}$ dan $\boldsymbol{v \in FV(P)}$, dimana $\boldsymbol{P \equiv (\lambda x.x)(vu)}$ (000). Dalam kasus II, pertama kita harus menerapkan algoritma PT pada P, untuk mengecek apakah P *typable* atau tidak. Hal ini dijelaskan pada tahap 4.

4. Untuk *term* $\boldsymbol{(000) \equiv (\lambda x.x)(vu)}$, terapkan kasus IV. Dalam kasus IV, pertama kita harus menerapkan algoritma PT pada P dan Q (dimana $\boldsymbol{P \equiv (\lambda x.x)}$ (0001)

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

dan $\boldsymbol{Q} \equiv (\boldsymbol{vu})$ (0002)), untuk mengecek apakah P dan Q *typable* atau tidak. Hal ini dijelaskan pada tahap 5 dan 6.

5. Untuk *term* $\mathbf{(0001)} \equiv (\boldsymbol{\lambda x.x})$, terapkan kasus II, dengan $\boldsymbol{M} \equiv \boldsymbol{\lambda x.x}$ dan $\boldsymbol{x} \in \boldsymbol{FV(P)}$, dimana $\boldsymbol{P} \equiv \boldsymbol{x}$ (00010). Dalam kasus II, pertama kita harus menerapkan algoritma PT pada P, untuk mengecek apakah P *typable* atau tidak. Hal ini dijelaskan pada tahap 7.

6. Untuk *term* $\mathbf{(0002)} \equiv \boldsymbol{vu}$, terapkan kasus IV. Dalam kasus IV, pertama kita harus menerapkan algoritma PT pada P dan Q (dimana $\boldsymbol{P} \equiv \boldsymbol{v}$ (00021) dan $\boldsymbol{Q} \equiv \boldsymbol{u}$ (00022)), untuk mengecek apakah P dan Q *typable* atau tidak. Hal ini dijelaskan pada tahap 8 dan 9.

7. Untuk *term* $\mathbf{(00010)} \equiv \boldsymbol{x}$, terapkan kasus I. Kemudian buat deduksi $\boldsymbol{\nabla}_{\mathbf{00010}}$ sebagai berikut:

$$\boldsymbol{x : a \mapsto x : a}$$

.

8. Untuk *term* $\mathbf{(00021)} \equiv \boldsymbol{v}$, terapkan kasus I. Kemudian buat deduksi $\boldsymbol{\nabla}_{\mathbf{00021}}$ sebagai berikut:

$$\boldsymbol{v : c \mapsto v : c}$$

.

9. Untuk *term* $\mathbf{(00022)} \equiv \boldsymbol{u}$, terapkan kasus I. Kemudian buat deduksi $\boldsymbol{\nabla}_{\mathbf{00022}}$ sebagai berikut:

$$\boldsymbol{u : b \mapsto u : b}$$

.

10. Sekarang kembali ke tahap 5 (Melanjutkan penerapan kasus II pada 0001). Karena (00010) *typable* dan memiliki *principal deduction* $\boldsymbol{\nabla}_{\mathbf{00010}}$ dengan konklusi

$$\boldsymbol{x : a \mapsto x : a}.$$

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

Terapkan aturan $(\rightarrow I)_{main}$ untuk membuat deduksi ∇_{0001} yang konklusinya adalah sebagai berikut:

$$\mapsto \lambda x.x : a \rightarrow a$$

.

11. Sekarang kembali ke tahap 6 (Melanjutkan penerapan kasus IV pada 0002). Ternyata konklusi ∇_{00021} dan ∇_{00022} adalah sebagai berikut:

$$v : c \mapsto v : c$$
$$u : b \mapsto u : b$$

Karena PT(00021) atomik, lanjutkan ke subkasus IVb. Kemudian pilih sembarang *variable*, misalkan d, yang berlaku $d \notin Vars(\nabla_{00021}) \cup \nabla_{00022}$ dan terapkan algoritma unifikasi pada pasangan $\langle c \rangle, \langle b \rightarrow d \rangle$ (kita sebut pasangan (1)). Ternyata pasangan (1) memiliki *unifier* u (dari hasil penerapan algoritma unifikasi pada pasangan (1)), yaitu $[b \rightarrow d/c]$. Oleh karena itu, selanjutnya terapkan subsubkasus IVb2. Lakukan *renaming* jika diperlukan untuk memastikan bahwa:

$Dom(u) = Vars(c, b \rightarrow d)$, dan

$Range(u) \cap V = \emptyset$,

dimana

$V = (Vars(\nabla_{00021}) \cup Vars(\nabla_{00022})) - Dom(u)$. Kemudian terapkan u pada ∇_{00021} dan ∇_{00022}, berdasarkan definisi u, kita dapat

$v : b \rightarrow d \mapsto v : b \rightarrow d$

$u : b \mapsto u : b$.

Selanjutnya buat deduksi ∇_{0002} dengan menerapkan aturan $(\rightarrow E)$ pada $u(\nabla_{00021})$ dan $u(\nabla_{00022})$. Konklusi deduksi ∇_{0002} adalah sebagai berikut:

$$v : b \rightarrow d, u : b \mapsto vu : d$$

12. Sekarang kembali ke tahap 4 (Melanjutkan penerapan kasus IV pada 000). Ternyata konklusi ∇_{0001} dan ∇_{0002} adalah sebagai berikut:

$\mapsto \lambda x.x : a \rightarrow a$

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

$\boldsymbol{v : b \to d, u : b \mapsto vu : d}$

Karena PT(0001) komposit, lanjutkan ke subkasus IVa. Kemudian terapkan algoritma unifikasi pada pasangan $\boldsymbol{\langle a \rangle, \langle d \rangle}$ (kita sebut pasangan (2)). Ternyata pasangan (2) memiliki *unifier* $\boldsymbol{u}$ (dari hasil penerapan algoritma unifikasi pada pasangan (2)), yaitu $[\boldsymbol{a/d}]$. Oleh karena itu, selanjutnya terapkan subsubkasus IVa2. Lakukan *renaming* jika diperlukan untuk memastikan bahwa:

$\boldsymbol{Dom(u) = Vars(a, d)}$, dan

$\boldsymbol{Range(u) \cap V = \emptyset}$,

dimana

$\boldsymbol{V = (Vars(\nabla_{0001}) \cup Vars(\nabla_{0002})) - Dom(u)}$. Kemudian terapkan $\boldsymbol{u}$ pada $\boldsymbol{\nabla_{0001}}$ dan $\boldsymbol{\nabla_{0002}}$, berdasarkan definisi $\boldsymbol{u}$, kita dapat

$\boldsymbol{\mapsto \lambda x.x : a \to a}$

$\boldsymbol{v : b \to a, u : b \mapsto vu : a}$

Selanjutnya buat deduksi $\boldsymbol{\nabla_{000}}$ dengan menerapkan aturan $\boldsymbol{(\to\ E)}$ pada $\boldsymbol{u(\nabla_{0001})}$ dan $\boldsymbol{u(\nabla_{0002})}$. Konklusi deduksi $\boldsymbol{\nabla_{000}}$ adalah sebagai berikut:

$$\boldsymbol{v : b \to a, u : b \mapsto (\lambda x.x)(vu) : a}$$

13. Sekarang kembali ke tahap 3 (Melanjutkan penerapan kasus II pada 00). Karena (000) *typable* dan memiliki *principal deduction* $\boldsymbol{\nabla_{000}}$ dengan konklusi

$$\boldsymbol{v : b \to a, u : b \mapsto (\lambda x.x)(vu) : a}.$$

Terapkan aturan $\boldsymbol{(\to I)_{main}}$ untuk membuat deduksi $\boldsymbol{\nabla_{00}}$, yang konklusinya adalah sebagai berikut:

$$\boldsymbol{u : b \mapsto \lambda v.(\lambda x.x)(vu) : (b \to a) \to a}$$

.

14. Sekarang kembali ke tahap 2 (Melanjutkan penerapan kasus II pada 0). Karena (00) *typable* dan memiliki *principal deduction* $\boldsymbol{\nabla_{00}}$ dengan konklusi

$$\boldsymbol{u : b \mapsto \lambda v.(\lambda x.x)(vu) : (b \to a) \to a}.$$

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

Terapkan aturan $(\rightarrow \boldsymbol{I})_{main}$ untuk membuat deduksi $\boldsymbol{\nabla_0}$, yang konklusinya adalah sebagai berikut:

$$\mapsto \boldsymbol{\lambda uv.(\lambda x.x)(vu) : b \rightarrow (b \rightarrow a) \rightarrow a}$$

.

15. Sekarang kembali ke tahap 1 (Melanjutkan penerapan kasus III pada $\boldsymbol{M}$, dimana $\boldsymbol{M \equiv \lambda yuv.(\lambda x.x)(vu)}$). Karena (0) *typable* dan memiliki *principal deduction* $\boldsymbol{\nabla_0}$ dengan konklusi

$$\mapsto \boldsymbol{\lambda uv.(\lambda x.x)(vu) : b \rightarrow (b \rightarrow a) \rightarrow a}.$$

Pilih sembarang *variable* yang tidak berada pada $\boldsymbol{\nabla_0}$, misalkan $\boldsymbol{c}$. Lalu terapkan aturan $(\rightarrow \boldsymbol{I})_{vac}$ untuk membuat deduksi $\boldsymbol{\nabla_M}$, yang konklusinya adalah sebagai berikut:

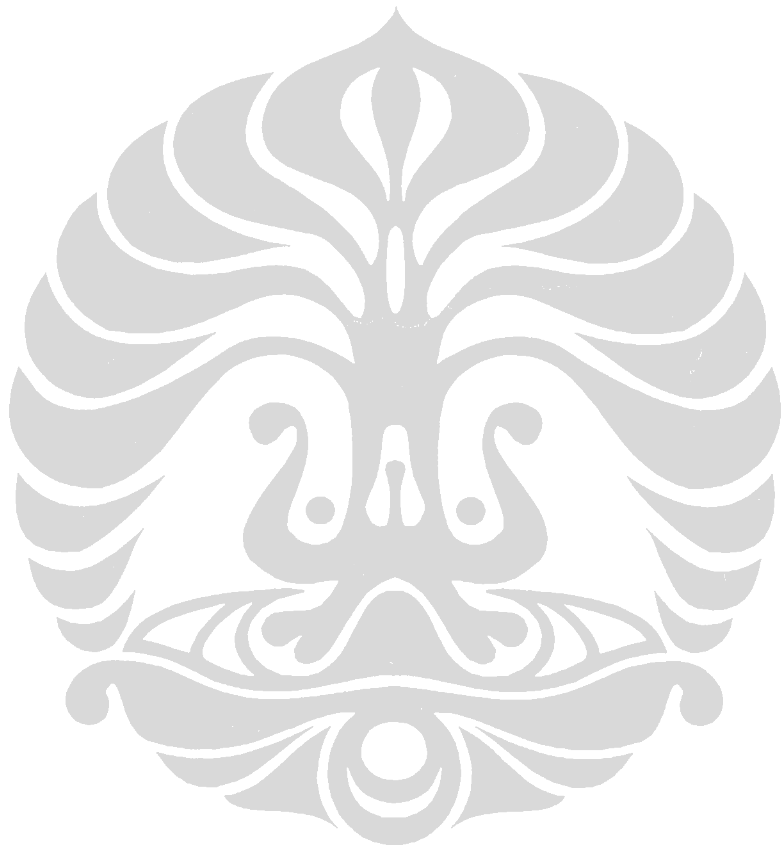
$$\mapsto \boldsymbol{\lambda yuv.(\lambda x.x)(vu) : c \rightarrow b \rightarrow (b \rightarrow a) \rightarrow a}$$

.

Berikut adalah rangkuman deduksi hasil penerapan algoritma PT terhadap $\boldsymbol{\lambda yuv.(\lambda x.x)(vu)}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x : a \mapsto x : a}{\mapsto \lambda x.x : a \rightarrow a}(\rightarrow I) \qquad \dfrac{v : b \rightarrow a \mapsto v : b \rightarrow a \qquad u : b \mapsto u : b}{v : b \rightarrow a, u : b \mapsto vu : a}(\rightarrow E)}{v : b \rightarrow a, u : b \mapsto (\lambda x.x)(vu) : a}(\rightarrow E)}{u : b \mapsto \lambda v.(\lambda x.x)(vu) : (b \rightarrow a) \rightarrow a}(\rightarrow I)}{\mapsto \lambda uv.(\lambda x.x)(vu) : b \rightarrow (b \rightarrow a) \rightarrow a}(\rightarrow I)}{\mapsto \lambda yuv.(\lambda x.x)(vu) : c \rightarrow b \rightarrow (b \rightarrow a) \rightarrow a}(\rightarrow I)$$

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008



Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

Bab 6

Implementasi Algoritma *Principal Type*

Bab ini memaparkan implementasi algoritma *Principal Type*, yaitu algoritma untuk memberikan tipe pada sebuah **λ**-*term* yang dijelaskan pada bab 5. Penjelasan pada bab ini dimulai dengan pemaparan mengenai bentuk representasi data dalam program. Kemudian bab ini dilanjutkan dengan penjelasan alur kerja program, dan parser untuk **λ**-*term*. Setelah itu, penjelasan dilanjutkan dengan implementasi dari algoritma Unifikasi (*Unification Algorithm*). Terakhir, bab ini ditutup dengan pemaparan tentang implementasi *Principal Type Algorithm* dan pengujian hasil implementasi.

6.1 Bentuk Representasi Data dalam Program

Sub-bab ini menjelaskan bagaimana data direpresentasikan dalam program. Adapun beberapa data yang perlu direpresentasikan dalam program adalah **λ**-*Term* (*abstraction*, *application*, dan *term-variable*), tipe (*composite type* dan *type-variable*), substitusi tipe, *type-assignment* ($TA_{\boldsymbol{\lambda}}$), *type-context*($\boldsymbol{\Gamma}$), dan $TA_{\boldsymbol{\lambda}}$*-Formula*.

6.1.1 Representasi λ-*Term* dalam Program

Sebuah **λ**-*term* direpresentasikan dalam bentuk *tree* (seperti pohon konstruksi **λ**-*term* yang dijelaskan pada sub-bab 3.2.6). Ada 3 *functor* yang digunakan untuk merepresentasikan *node* dalam *tree*, yaitu:

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

1. `termVar /1`, merepresentasikan *term-variable.* Adapun struktur *functor* tersebut adalah sebagai berikut:

   `termVar([Nama` ***`term-variable`***`]).`

   Dalam program, *term-variable* yang dapat diterima tersusun dari satu huruf (u, v, w, x, y, atau z) yang diikuti atau tidak diikuti dengan *subscript* angka. Kemudian di dalam *tree* dari $\boldsymbol{\lambda}$-*term* tersebut, *functor* `termVar /1` selalu menjadi *leaves.*

2. `abstraction /2`, merepresentasikan sebuah abstraksi dari $\boldsymbol{\lambda}$-*term.* Adapun struktur *functor* tersebut adalah sebagai berikut:

   `abstraction([`***`term-variable`***`], [`***`body`***` dari abstraksi]).`

   Atau

   `abstraction(termVar([Nama` ***`term-variable`***`]), [`$\boldsymbol{\lambda}$***`-term`***`]).`

   Sebuah abstraksi dari $\boldsymbol{\lambda}$-*term* terdiri dari *abstractor* dan *body.* Misalnya untuk *term* $\boldsymbol{\lambda x.yx}$, *abstractor*-nya adalah $\boldsymbol{\lambda x}$ dan *body*-nya adalah $\boldsymbol{yx}$. Oleh karena itu, struktur *functor* `abstraction` /2 terdiri dari 2 argumen, yaitu *term-variable* dan *body* dari abstraksi yang merupakan sebuah $\boldsymbol{\lambda}$-*term.*

3. `application /2`, merepresentasikan aplikasi dari 2 $\boldsymbol{\lambda}$-*term.* Adapun struktur *functor* tersebut adalah sebagai berikut:

   `application([`$\boldsymbol{\lambda}$***`-term`***`], [`$\boldsymbol{\lambda}$***`-term`***`]).`

   Sebuah aplikasi $\boldsymbol{\lambda}$-*term* terdiri dari dua buah $\boldsymbol{\lambda}$-*term.* Oleh karena itu, struktur *functor* `application` /2 terdiri dari 2 argumen, yaitu dua buah $\boldsymbol{\lambda}$-*term.*

Berikut adalah contoh representasi $\boldsymbol{\lambda}$-*term* dalam bentuk *tree* yang disusun dari *functor-functor* tersebut:

- $\boldsymbol{x}$, direpresentasikan menjadi
  `termVar(x).`

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

- $\boldsymbol{\lambda x.y}$, direpresentasikan menjadi

  ```
  abstraction(termVar(x), termVar(y)).
  ```

- $\boldsymbol{xy}$, direpresentasikan menjadi

  ```
  application(termVar(x), termVar(y)).
  ```

- $\boldsymbol{\lambda x.xy}$, direpresentasikan menjadi

  ```
  abstraction(termVar(x), application(termVar(x), termVar(y))).
  ```

- $\boldsymbol{\lambda v.\lambda x.xy}$, direpresentasikan menjadi

  ```
  abstraction(termVar(v), abstraction(termVar(x),
        application(termVar(x), termVar(y)))).
  ```

6.1.2 Representasi Tipe dalam Program

Tipe dari $\boldsymbol{\lambda}$-*term* memiliki dua jenis bentuk, yaitu sebagai *type-variable* dan sebagai tipe komposit. Sebagai tipe komposit, sebuah tipe memiliki bentuk umum sebagai berikut:

$$[\boldsymbol{Tipe}] \rightarrow [\boldsymbol{Tipe}],$$

dan contoh konkritnya adalah sebagai berikut:

$$\boldsymbol{\rho} \rightarrow \boldsymbol{\tau},$$

dengan $\boldsymbol{\rho}$ dan $\boldsymbol{\tau}$ adalah sebuah tipe. Oleh karena itu $\boldsymbol{\rho}$ dan $\boldsymbol{\tau}$ dapat berupa *type-variable* ataupun tipe komposit. Dalam program, sebuah tipe direpresentasikan dalam bentuk *tree*. Di dalam *tree* tersebut terdapat 2 *functor* yang digunakan untuk merepresentasikan *node*, yaitu:

1. `tpV /1`. *Functor* ini merepresentasikan *type-variable*. Adapun struktur *functor* tersebut adalah sebagai berikut:

   ```
   tpV([Nama type-variable]).
   ```

   Functor ini selalu menjadi *leaves* dari *tree* yang merepresentasikan tipe.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

2. `tp /2`. *Functor* ini merepresentasikan sebuah tipe komposit. Adapun struktur *functor* tersebut adalah sebagai berikut:

```
tp([Tipe], [Tipe]).
```

Sebuah tipe komposit terdiri dari dua buah tipe. Oleh karena itu, struktur *functor* `tp` /2 terdiri dari 2 argumen, yaitu dua buah tipe.

Berikut adalah contoh representasi tipe dalam bentuk *tree* yang disusun dari *functor* tersebut:

- $\boldsymbol{a}$, direpresentasikan menjadi
  `tpV(a).`
- $\boldsymbol{a \rightarrow b}$, direpresentasikan menjadi
  `tp(tpV(a), tpV(b)).`
- $\boldsymbol{c \rightarrow (a \rightarrow b)}$, direpresentasikan menjadi
  `tp(tpV(c), tp(tpV(a), tpV(b))).`
- $\boldsymbol{(a \rightarrow b) \rightarrow (c \rightarrow d)}$, direpresentasikan menjadi
  `tp(tp(tpV(a), tpV(b)), tp(tpV(c), tpV(d))).`

6.1.3 Representasi substitusi Tipe (*Type-Substitution*) dalam Program

Bentuk substitusi tipe adalah sebagai berikut:

$$[\boldsymbol{\sigma_1/a_1, \ldots, \sigma_n/a_n}],$$

dengan $\boldsymbol{a_1, \ldots, a_n}$ merupakan *type-variable* yang berbeda, dan $\boldsymbol{\sigma_1, \ldots, \sigma_n}$ merupakan sembarang tipe. Di dalam program, sebuah substitusi tipe direpresentasikan dengan *functor* `s /2`, yang strukturnya sebagai berikut:

```
s([Tipe], [type-variable]).
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

dimana penulisan (representasi) [Tipe] dan [*type-variable*] dalam program mengacu pada penjelasan representasi tipe dan *type-variable* dalam program yang dijelaskan pada sub-bab 6.1.2. Berikut adalah contoh representasi sebuah substitusi tipe dalam program:

- $[\boldsymbol{a}/\boldsymbol{c}]$, direpresentasikan menjadi `s(tpV(a), tpV(c))`.
- $[\boldsymbol{a} \rightarrow \boldsymbol{b}/\boldsymbol{c}]$, direpresentasikan menjadi `s(tp(tpV(a), tpV(b)), tpV(c))`.

Kemudian substitusi tipe direpresentasikan dalam sebuah *list*, yang strukturnya adalah seperti berikut ini:

```
[s([Tipe], [type-variable]), s([Tipe], [type-variable]), ...]
```

Contoh:

$$[\ \boldsymbol{a}/\boldsymbol{c},\ (\boldsymbol{a} \rightarrow \boldsymbol{b})/\boldsymbol{d}]$$

direpresentasikan menjadi:

```
[s(tpV(a), tpV(c)), s(tp(tpV(a), tpV(b)), tpV(d))]
```

6.1.4 Representasi *Type-Assignment* dalam Program

Struktur dari *type-assignment* adalah sebagai berikut:

$$\boldsymbol{M} : \boldsymbol{\tau}$$

dimana M adalah sebuah $\boldsymbol{\lambda}$-*term* dan $\boldsymbol{\tau}$ adalah sembarang tipe. Di dalam program, sebuah *type-assignment* direpresentasikan dengan *functor* `ta /2`. Adapun strukturnya adalah sebagai berikut:

```
ta([λ-term], [Tipe])
```

dimana struktur [$\boldsymbol{\lambda}$-*term*] mengacu pada penjelasan struktur $\boldsymbol{\lambda}$-*term* pada sub-bab 6.1.1, dan struktur representasi [Tipe] mengacu pada penjelasan struktur $\boldsymbol{\lambda}$-*term* pada sub-bab 6.1.2. Berikut adalah contoh representasi *type-assignment* dalam program:

- $\boldsymbol{x} : \boldsymbol{a}$, direpresentasikan menjadi
  `ta(termVar(x), tpV(a))`.
- $\boldsymbol{\lambda x.x} : \boldsymbol{a} \rightarrow \boldsymbol{a}$, direpresentasikan menjadi
  `ta(abstraction(termVar(x), termVar(x)), tp(tpV(a), tpV(a)))`.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

6.1.5 Representasi *Type-Context*(Γ) dalam Program

Sebuah *type-context* **Γ** merupakan himpunan *type-assignment* yang jumlahnya berhingga (mungkin kosong).

$$\Gamma = \{x_1 : \rho_1, \ldots, x_m : \rho_m\}.$$

Dalam program, *type-context* direpresentasikan dengan sebuah *list* yang terdiri dari sejumlah *type-assignment* (direpresentasikan oleh *functor* `ta /2`, seperti yang dijelaskan pada sub-bab 6.1.4). Berikut adalah contoh representasi sebuah *type-context* dalam program:

- $\Gamma = \{x : \rho, y : \tau\}$,
  direpresentasikan menjadi `[ta(termVar(x), tpV(`ρ`)), ta(termVar(y), tpV(`τ`))]`
- $\Gamma = \{\}$,
  direpresentasikan menjadi `[ ]`

6.1.6 Representasi *TA*$_\lambda$*-Formula* dalam Program

Sebuah *TA*$_\lambda$*-Formula*, memiliki bentuk

$$\Gamma \mapsto M : \tau$$

$$[\textit{type-context}] \mapsto [\textit{type-assignment}]$$

Dalam program, sebuah *TA*$_\lambda$*-Formula* direpresentasikan dengan *functor* `ded /2`. Adapun struktur *functor* tersebut adalah sebagai berikut:

```
ded([type-context], [type-assignment])
```

dimana struktur [*type-context*] mengacu pada sub-bab 6.1.5, dan [*type-assignment*] mengacu pada sub-bab 6.1.4. Berikut adalah contoh representasi sebuah *TA*$_\lambda$*-Formula* dalam program:

- $x : a \mapsto x : a$,
  direpresentasikan menjadi
  `ded([ta(termVar(x), tpV(a))], ta(termVar(x), tpV(a)))`

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

- $\boldsymbol{y : c \rightarrow a, z : c \mapsto yz : a}$,

  direpresentasikan menjadi

  ```
  ded([ta(termVar(y), tp(tpV(c),tpV(a)))],
    ta(application(termVar(y), termVar(z)), tpV(a)))
  ```

6.1.7 Representasi *Deduction Tree* dalam Program

Dalam program terdapat 3 jenis *Deduction Tree*, masing-masing direpresentasikan sebagai berikut:

- *Deduction tree* untuk deduksi *term variable* direpresentasikan oleh *functor* `termVarDedTree /1`. Struktur *functor* tersebut adalah sebagai berikut:

  `termVarDedTree([formula TA`$_\lambda$`]).`

- *Deduction tree* untuk deduksi sebuah abstraksi $\boldsymbol{\lambda x.P}$ direpresentasikan oleh *functor* `abstractionDedTree /2`. Struktur *functor* tersebut adalah sebagai berikut:

  `abstractionDedTree([formula TA`$_\lambda$`], [`*deduction tree* $\boldsymbol{P}$`]).`

  Atau

  `abstractionDedTree([Konklusi deduksi` $\boldsymbol{\lambda x.P}$`], [`*deduction tree* $\boldsymbol{P}$`]).`

- *Deduction tree* untuk deduksi sebuah aplikasi $\boldsymbol{PQ}$ direpresentasikan oleh *functor* `abstractionDedTree /2`. Struktur *functor* tersebut adalah sebagai berikut:

  `applicationDedTree([formula TA`$_\lambda$`],`
  `[`*deduction tree* $\boldsymbol{P}$`],[`*deduction tree* $\boldsymbol{Q}$`]).`

  Atau

  `applicationDedTree([Konklusi deduksi` $\boldsymbol{PQ}$`],`
  `[`*deduction tree* $\boldsymbol{P}$`],[`*deduction tree* $\boldsymbol{Q}$`]).`

6.2 Antarmuka Program

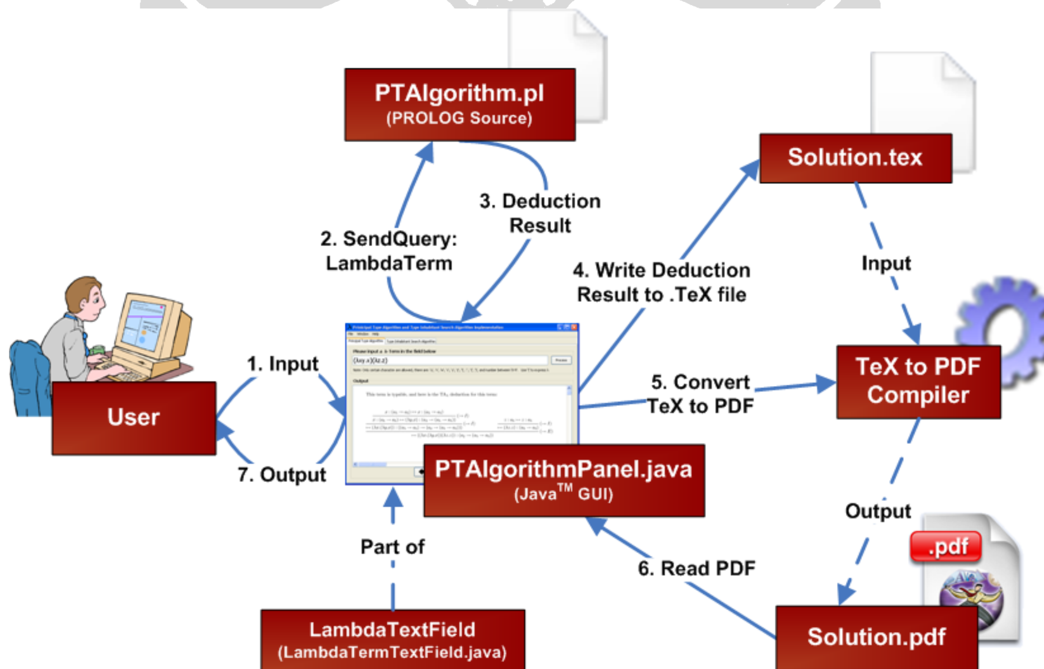
Antarmuka program ini dirancang untuk memudahkan pengguna dalam menggunakan program implementasi algoritma ini. Cara yang dilakukan ialah dengan meminimalisasi

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

kemungkinan kesalahan *input* dari pengguna dan menggunakan simbol yang representatif. Selain itu, dalam merancang antarmuka ini juga digunakan beberapa aturan 8 *Golden Rules* dalam mendesain *user interface* [BS05], yaitu pencegahan kesalahan, pengurangan beban ingatan jangka pendek pengguna, mempertahankan konsistensi, memberikan umpan balik yang informatif, dan menjadikan pengguna sebagai inisiator dari aktifitas, bukan sebagai responden. *Screen shot* aplikasi ini dapat dilihat pada Lampiran A.

6.3 Alur Kerja Program

Sub-bab ini menjelaskan alur kerja program secara umum, yaitu dari saat membaca *input*, memproses *input*, dan terakhir adalah mengeluarkan (mem-*print*) *output*. Alur kerja ini digambarkan pada Gambar 6.1, dan berikut ini adalah penjelasannya:



Gambar 6.1: Alur kerja program implementasi algoritma PT.

1. Penguna memberikan *input* $\boldsymbol{\lambda}$-*term* pada LambdaTermTextField (`LambdaTermTextField.java`).

2. `PTAlgorithmPanel.java` mengirimkan *query* ke PROLOG (`PTAlgorithm.pl`) untuk mencari tipe dari $\boldsymbol{\lambda}$-*term* yang diberikan.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

3. `PTAlgorithm.pl` mengirimkan *output* hasil penerapan algoritma PT ke `PTAlgorithmPanel.java`. Output ini dalam format penulisan LaTeX.

4. `PTAlgorithmPanel.java` menuliskan hasil yang diberikan `PTAlgorithm.pl` ke dalam TeX file (Solution.tex).

5. `PTAlgorithmPanel.java` meng-*compile* `Deduction.tex` menjadi file PDF menggunakan TeX to PDF *compiler*. Hasilnya adalah `Solution.pdf`.

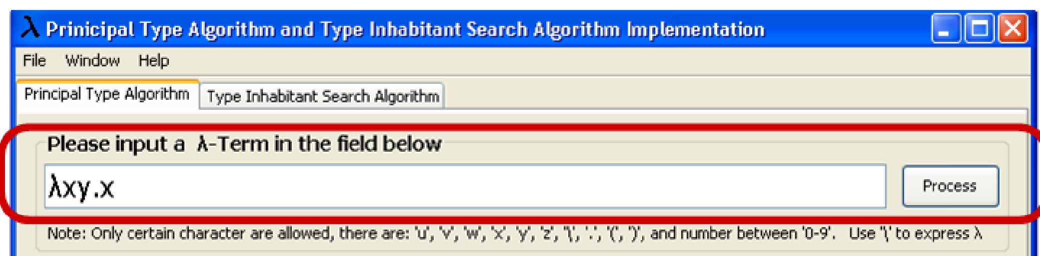
6. `PTAlgorithmPanel.java` membaca `Solution.pdf`.

7. `PTAlgorithmPanel.java` menampilkan *output* hasil deduksi terhadap *term* yang diberikan kepada pengguna.

Penjelasan lebih detail tentang implementasi untuk bagian *input*, pemrosesan dan *output* disampaikan pada sub-bab berikutnya, yaitu sub-bab 6.3.1, 6.3.2 dan 6.3.3.

6.3.1 Bagian *Input*

Sub-bab ini menjelaskan bagian program yang menangani *input* dari pengguna. Proses ini ditangani oleh `PTAlgorithmPanel.java` dan `LambdaTermTextField.java`. `LambdaTermTextField.java` merupakan komponen *textfield* khusus yang dibuat dengan meng-*extends* `JTextField`. `LambdaTermTextField` merupakan *textfield* yang hanya memperbolehkan karakter tertentu untuk dimasukkan, yaitu: 'u', 'v', 'w', 'x', 'y', 'z', '\', '(', ')', '.', '1', '2', '3', '4', '5', '6', '7', '8', '9', '0'. Selain itu, `LambdaTermTextField` juga akan menampilkan karakter '\' sebagai $\boldsymbol{\lambda}$. Gambar 6.2 menunjukkan *screenshot* dari `LambdaTermTextField` tersebut.



Gambar 6.2: Gambar $\boldsymbol{\lambda}$-*term textfield*.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

6.3.2 Bagian Pemrosesan

Ketika pengguna menekan tombol proses, `PTAlgorithmPanel.java` akan membuat sebuah *List* yang nantinya menjadi *input* dari *parser* **λ**-*term* pada `PTAlgorithm.pl`, dan kemudian mengirimkan *query* ke `PTAlgorithm.pl`. *Query* tersebut memanggil predikat `applyPTA /1`. Predikat inilah yang melakukan pemrosesan *input*. Dalam predikat ini dilakukan *parsing* terhadap *input* untuk membentuk *tree* dari *term* yang diberikan, yang kemudian menjadi *input* dari algoritma PT. Adapun implementasi predikat tersebut adalah sebagai berikut:

```
applyPTA(LambdaTermInListFormat) :-
    reset_gensym(a),
    lambdaTerm(Tree,LambdaTermInListFormat,[]), !,
    pta(Tree,_,Result,_,DedTree),
    write(Result), write(';\n'),
    branchOnDetermineTypableType(Result, DedTree).
applyPTA(_) :-
    write('invalid_input;').
```

Dalam `applyPTA /1` terdapat satu argumen yaitu [LambdaTermInListFormat]. [LambdaTermInListFormat] merupakan *input* **λ**-*term* yang diberikan dalam bentuk *List*. Selain itu terdapat juga pemanggian terhadap predikat `branchOnDetermineTypableType /2`. Predikat tersebut berfungsi untuk pengaturan *output*, penjelasan lengkapnya dijelaskan pada sub-bab 6.3.3.

6.3.3 Bagian *Output*

Sub-bab ini menjelaskan bagian program yang menangani *output* hasil pemrosesan. Ada tiga jenis *output* dari program ini, yaitu:

1. Saat *term* yang diberikan tidak valid, program mengeluarkan *output* yang memberikan informasi bahwa *term* yang diberikan tidak valid.

2. Saat *term* yang diberikan valid, namun tidak dapat diberikan tipe (*untypable*), program mengeluarkan *output* yang memberikan informasi bahwa *term* yang diberikan tidak dapat diberikan tipe (*untypable*).

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

3. Saat *term* yang diberikan valid dan dapat diberikan tipe (*typable*), program mengeluarkan pernyataan bahwa *term* yang diberikan `'typable'`, kemudian program juga mengeluarkan deduksi TA_{λ} dari *term* tersebut.

Dalam bagian *output* ini, terdapat predikat dalam `PTAlgorithm.pl` yang berperan, yaitu:

`branchOnDetermineTypableType /2.`

Predikat inilah yang menentukan apakah program harus mengeluarkan deduksi TA_{λ} dari *term* yang diberikan atau tidak. Adapun berikut ini adalah implementasi dari predikat `branchOnDetermineTypableType /2`:

```
branchOnDetermineTypableType(Typable,DedTree) :-
    Typable == 'typable', !,
    generateStringForDeductionTree(DedTree,DedTreeStr),
    write(DedTreeStr).
branchOnDetermineTypableType(Typable, _) :-
    Typable == 'not_typable', !.
```

Selain itu, terdapat juga predikat yang berfungsi untuk mentranslasikan bentuk *output* dari bentuk struktur data yang direpresentasikan dengan *functor* (seperti yang dijelaskan pada sub-bab 6.1) ke bentuk dalam format LaTeX . Predikat tersebut adalah:

- `generateStringForDeductionTree /2`. Mentranslasikan deduksi TA_{λ} yang direpresentasikan dalam bentuk *tree*, menjadi bentuk tulisan (*string*) dalam format LaTeX. Berikut adalah implementasi predikat tersebut:

```
generateStringForDeductionTree(termVarDedTree(Ded_M),
                                        DedTreeStr) :- !,
    generateStringForDeduction2(Ded_M, DedTreeStr).
generateStringForDeductionTree(applicationDedTree(Ded_PQ,
                                               PDedVarTree, QDedVarTree),
                                        StrResult) :- !,
    generateStringForDeductionTree(PDedVarTree, PDedVarTreeStr),
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
      generateStringForDeductionTree(QDedVarTree, QDedVarTreeStr),
      generateStringForDeduction2(Ded_PQ, Ded_PQStr),
      atom_concat('\\prooftree ','\n',A),
      atom_concat(A,PDedVarTreeStr,B),
      atom_concat(B,'\\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \n',C),
      atom_concat(C,QDedVarTreeStr,D),
      atom_concat(D,'\n\\justifies \n',E),
      atom_concat(E,Ded_PQStr,F),
      atom_concat(F,'\n\\using \n',G),
      atom_concat(G,'(\\rightarrow E) \n',H),
      atom_concat(H,'\\endprooftree',StrResult).
   generateStringForDeductionTree(abstractionDedTree(Ded_LP,
                                                     BodyDedTree),
                                  StrResult) :- !,
      generateStringForDeductionTree(BodyDedTree, BodyDedTreeStr),
      generateStringForDeduction2(Ded_LP, Ded_LPStr),
      atom_concat('\\prooftree ','\n',A),
      atom_concat(A,BodyDedTreeStr,B),
      atom_concat(B,'\n\\justifies \n',C),
      atom_concat(C,Ded_LPStr,D),
      atom_concat(D,'\n\\using \n',E),
      atom_concat(E,'(\\rightarrow I) \n',F),
      atom_concat(F,'\\endprooftree',StrResult).
```

- `generateStringForSeqDeduction2 /2`. Mentranslasikan *list* dari deduksi yang direpresentasikan dalam bentuk *list* dari *functor* `ded /2`, menjadi bentuk tulisan (*string*) dalam format LaTeX. Berikut adalah implementasi predikat tersebut:

```
   generateStringForSeqDeduction2([], '') :-  !.
   generateStringForSeqDeduction2([DedH|[]], StrResult) :- !,
      generateStringForDeduction2(DedH, Str),
      atom_concat(Str,'\n\n\t',StrResult).
   generateStringForSeqDeduction2([DedH|DedT], StrResult) :- !,
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
      generateStringForSeqDeduction2(DedT,SubResult),
      generateStringForDeduction2(DedH, Str),
      atom_concat(Str,'\n\n\t',Str2),
      atom_concat(SubResult, Str2, StrResult).
```

- `generateStringForDeduction2 /2`. Mentranslasikan sebuah deduksi yang direpresentasikan dalam *functor* `ded /2`, menjadi bentuk tulisan (*string*) dalam format LaTeX. Berikut adalah implementasi predikat tersebut:

```
generateStringForDeduction2(ded([],[]),'') :- !.
generateStringForDeduction2(ded(Context,TA),
                              StrResult) :- !,
   generateStringForTA2(TA,StrTA),
   generateStringForContext2(Context,StrCon),
   atom_concat(StrCon,' \\mapsto ',A),
   atom_concat(A,' ',B),
   atom_concat(B,StrTA,StrResult).
```

- `generateStringForContext2 /2`. Mentranslasikan sebuah *type-context* yang direpresentasikan dalam bentuk *list* dari *type-assignment*, menjadi bentuk tulisan (*string*) dalam format LaTeX. Berikut adalah implementasi predikat tersebut:

```
generateStringForContext2([], '') :- !.
generateStringForContext2([HCntx | []], StrHCntx) :-!,
   generateStringForTA2(HCntx,StrHCntx).
generateStringForContext2([HCntx|TCntx], StringResult) :- !,
   generateStringForTA2(HCntx,StrHCntx),
   generateStringForContext2(TCntx,StrTCntx),
   atom_concat(StrHCntx, ',', B),
   atom_concat(B , StrTCntx, C),
   atom_concat(C,' ',StringResult).
```

- `generateStringForTA2 /2`. Mentranslasikan sebuah *type-assignment* yang direpresentasikan dalam *functor* `ta /2`, menjadi bentuk tulisan (*string*) dalam format LaTeX. Berikut adalah implementasi predikat tersebut:

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
generateStringForTA2(ta(LambdaTerm,Type), StringResult) :- !,
   generateStringForLambdaTerm2(LambdaTerm,LTS),
   generateStringforType2(Type,TS),
   atom_concat(LTS,':',A),
   atom_concat(A,TS,StringResult).
```

- `generateStringForLambdaTerm2 /2`. Mentranslasikan sebuah *lambda-term* yang direpresentasikan dalam bentuk *tree* (seperti yang dijelaskan pada sub-bab 6.1.1), menjadi bentuk tulisan (*string*) dalam format LaTeX. Berikut adalah implementasi predikat tersebut:

```
generateStringForLambdaTerm2(termVar(X),X) :- !.
generateStringForLambdaTerm2(abstraction(X,Y),StringResult) :- !,
   generateStringForLambdaTerm2(X,XS),
   generateStringForLambdaTerm2(Y,YS),
   atom_concat('(',' \\lambda ',A), atom_concat(A,XS,B),
   atom_concat(B,'.',C),atom_concat(C,YS,D),
   atom_concat(D,')',StringResult).
generateStringForLambdaTerm2(application(X,Y),StringResult) :- !,
   generateStringForLambdaTerm2(X,XS),
   generateStringForLambdaTerm2(Y,YS),
   atom_concat('(',XS,A), atom_concat(A,YS,B),
   atom_concat(B,')',StringResult).
```

- `generateStringforType /2`. Mentranslasikan sebuah tipe yang direpresentasikan dalam bentuk *tree* (seperti yang dijelaskan pada sub-bab 6.1.2), menjadi bentuk tulisan (*string*) dalam format LaTeX. Berikut adalah implementasi predikat tersebut:

```
generateStringforType2(tpV(X), Str) :- !,
   atom_concat(a,VarNum,X),
   atom_concat('a', '_{', Str1),
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
        atom_concat(Str1, VarNum, Str2 ),
        atom_concat(Str2, '}', Str).
    generateStringforType2(tp(X,Y), StringResult) :- !,
        generateStringforType2(X, XS),
        generateStringforType2(Y,YS),
        atom_concat('(',XS, XSA),
        atom_concat(YS, ')', YSA),
        atom_concat(XSA,' \\rightarrow ',XSP),
        atom_concat(XSP,YSA,StringResult).
```

6.4 Implementasi *Parser* untuk $\boldsymbol{\lambda}$-*Term*

Sub-bab ini menjelaskan implementasi *parser* untuk $\boldsymbol{\lambda}$-*term*. *Parser* ini diimplementasikan dengan *Definite Clause Grammar*(DCG) dalam PROLOG. *Code* implementasi lengkap dari *parser* $\boldsymbol{\lambda}$-*term* tersebut dapat dilihat pada Lampiran B.

Selain berfungsi untuk menguraikan input $\boldsymbol{\lambda}$-*term* yang diberikan, yang kemudian diputuskan apakah input yang diberikan diterima atau ditolak, parser ini juga membuat struktur *tree* yang merepresentasikan $\boldsymbol{\lambda}$-*term* yang diberikan. *Parser* $\boldsymbol{\lambda}$-*term* ini dikembangkan berdasarkan definisi dan pengertian dari $\boldsymbol{\lambda}$-*term* (seperti yang dijelaskan pada sub-bab 3.2.1). Berdasarkan definisi $\boldsymbol{\lambda}$-*term* pada sub-bab 3.2.1, sebuah $\boldsymbol{\lambda}$-*term* dapat berupa *term-variable*, abstraksi dari $\boldsymbol{\lambda}$-*term*, dan aplikasi dari $\boldsymbol{\lambda}$-*term*. Dari pengertian ini, dibuatlah spesifikasi DCG yang menyatakan hal tersebut sebagai berikut:

- Sebuah *term-variable* merupakan $\boldsymbol{\lambda}$-*term*.

```
lambdaTerm(X) --> termVariable(X).
```

- Sebuah abstraksi $\boldsymbol{\lambda}$-*term* merupakan $\boldsymbol{\lambda}$-*term*.

```
lambdaTerm(X) --> ltAbstraction(X).
```

- Sebuah aplikasi $\boldsymbol{\lambda}$-*term* merupakan $\boldsymbol{\lambda}$-*term*.

```
lambdaTerm(X) --> ltApplication(X).
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

Dalam spesifikasi DCG tersebut, setiap simbol *non-terminal* memiliki argumen X. Argumen tersebut merupakan struktur *tree* yang merepresentasikan **λ**-*term* yang bersangkutan. Selanjutnya implementasi DCG ini dilanjutkan dengan spesifikasi untuk *term-variable*, abstraksi, dan aplikasi dari **λ**-*term*.

Untuk *term-variable*, spesifikasi DCG dibuat berdasarkan penjelasan tentang notasi *term-variable* pada sub-bab 3.2.2. Dalam sub-bab tersebut, disebutkan bahwa *term-variable* dinyatakan dengan huruf kecil u, v, w, x, y, atau z, dengan atau tanpa *subscript* angka. Dari penjelasan tersebut, dibuatlah spesifikasi DCG untuk *term-variable* sebagai berikut:

- Sebuah *term-variable* yang berada di dalam tanda kurung juga merupakan *term-variable*.

```
termVariable(TV) --> ['('], termVariable(TV), [')'].
```

- Sebuah *term-variable* direpresentasikan dengan huruf kecil u, v, w, x, y, atau z, tanpa *subscript* angka.

```
termVariable(termVar(u)) --> [u].
termVariable(termVar(v)) --> [v].
termVariable(termVar(w)) --> [w].
termVariable(termVar(x)) --> [x].
termVariable(termVar(y)) --> [y].
termVariable(termVar(z)) --> [z].
```

- Sebuah *term-variable* direpresentasikan dengan huruf kecil u, v, w, x, y, atau z, dengan *subscript* angka.

```
termVariable(termVar(TermVar)) -->
   [u], number(N), {atom_concat('u',N,TermVar)}.
termVariable(termVar(TermVar)) -->
   [v], number(N), {atom_concat('v',N,TermVar)}.
termVariable(termVar(TermVar)) -->
   [w], number(N), {atom_concat('w',N,TermVar)}.
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
termVariable(termVar(TermVar)) -->
   [x], number(N), {atom_concat('x',N,TermVar)}.
termVariable(termVar(TermVar)) -->
   [y], number(N), {atom_concat('y',N,TermVar)}.
termVariable(termVar(TermVar)) -->
   [z], number(N), {atom_concat('z',N,TermVar)}.
singleNumber('0') --> [0].
singleNumber('1') --> [1].
singleNumber('2') --> [2].
singleNumber('3') --> [3].
singleNumber('4') --> [4].
singleNumber('5') --> [5].
singleNumber('6') --> [6].
singleNumber('7') --> [7].
singleNumber('8') --> [8].
singleNumber('9') --> [9].
number(SN) --> singleNumber(SN).
number(NUM) --> singleNumber(SN),
   number(N), {atom_concat(SN,N,NUM)}.
```

Untuk abstraksi dari $\boldsymbol{\lambda}$-*term*. Sesuai dengan penjelasan pada sub-bab 3.2.1, sebuah $\boldsymbol{\lambda}$-*abstract* memiliki bentuk $(\boldsymbol{\lambda x.M})$, dengan $\boldsymbol{x}$ merupakan suatu *term-variable* dan $\boldsymbol{M}$ adalah suatu $\boldsymbol{\lambda}$-*term*. Berdasarkan penjelasan struktur abstraksi $\boldsymbol{\lambda}$-*term* tersebut, disusunlah spesifikasi DCG untuk abstraksi dari $\boldsymbol{\lambda}$-*term* sebagai berikut:

- Sebuah abstraksi dari $\boldsymbol{\lambda}$-*term* yang berada pada tanda kurung merupakan abstraksi dari $\boldsymbol{\lambda}$-*term* juga.

```
ltAbstraction(ABS) -->
   ['('], ltAbstraction(ABS), [')'].
```

- Sebuah abstraksi $\boldsymbol{\lambda}$-*term* memiliki bentuk $(\boldsymbol{\lambda x.M})$.

```
ltAbstraction(abstraction(TV,LT)) -->
   [\],termVariable(TV),['.'],lambdaTerm(LT).
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

- Selain struktur umum seperti yang telah dijelaskan pada poin sebelumnya, terkadang penulisan abstraktor dalam abstraksi **λ**-*term* sering disingkat (seperti yang dijelaskan pada sub-bab 3.2.2). Contohnya adalah $\boldsymbol{\lambda xyz.M \equiv (\lambda x.(\lambda y.(\lambda z.M)))}$. Oleh karena itu, perlu dibuat spesifikasi DCG untuk mengakomodasi kebutuhan tersebut. Sehingga *parser* yang dibuat dapat menerima jenis *input* seperti itu. Berikut ini adalah spesifikasi DCG untuk kebutuhan tersebut:

```
ltAbstraction(abstraction(TV,LTX)) -->
   [\],termVariable(TV),absVar(LTX,LT),['.'],lambdaTerm(LT).
absVar(abstraction(TV,LT),LT) --> termVariable(TV).
absVar(abstraction(TV,ABS),LT) -->
   termVariable(TV),absVar(ABS,LT).
```

Untuk aplikasi pada **λ**-*term*, spesifikasi DCG yang dibuat juga mengikuti penjelasan aplikasi pada **λ**-*term* yang dijelaskan pada sub-bab 3.2.1. Berikut ini adalah spesifikasi DCG untuk aplikasi pada **λ**-*term* tersebut:

- Sebuah aplikasi dari **λ**-*term* yang berada di dalam tanda kurung juga merupakan suatu aplikasi pada **λ**-*term*.

```
ltApplication(App) --> ['('], ltApplication(App), [')'].
```

- Selanjutnya, aplikasi pada **λ**-*term* didefinisikan sebagai urutan dari *term-variable*, abstraksi **λ**-*term*, aplikasi **λ**-*term*, ataupun kombinasinya.

```
ltApplication(AppRes) -->
   termVariable(Tv), ltApp(Tv,AppRes).
ltApplication(AppRes) -->
   ['('], ltAbstraction(Abs), [')'],ltApp(Abs,AppRes).
ltApplication(AppRes) -->
   ['('], ltApplication(App), [')'],ltApp(App,AppRes).
ltApp(Lt,AppRes) -->
   termVariable(Tv), {AppRes = application(Lt,Tv)}.
ltApp(Lt,AppRes) -->
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
    ltAbstraction(Abs), {AppRes = application(Lt,Abs)}.
ltApp(Lt,AppRes) -->
    ['('], ltAbstraction(Abs), [')'],
    {AppRes = application(Lt,Abs)}.
ltApp(Lt,AppRes) -->
    ['('], ltApplication(App), [')'],
    {AppRes = application(Lt,App)}.
ltApp(Lt,AppRes) -->
    termVariable(Tv),
    {Apl = application(Lt,Tv)}, ltApp(Apl,AppRes).
ltApp(Lt,AppRes) -->
    ['('], ltAbstraction(Abs), [')'],
    {Apl = application(Lt,Abs)}, ltApp(Apl,AppRes).
ltApp(Lt,AppRes) -->
    ['('], ltApplication(App), [')'],
    {Apl = application(Lt,App)}, ltApp(Apl,AppRes).
```

6.5 Predikat Pendukung untuk Implementasi Algoritma Unifikasi dan *Principal Type Algorithm*

Sub-bab ini menjelaskan predikat-predikat pendukung yang digunakan dalam implementasi algoritma unifikasi dan *principal type algorithm*. Beberapa predikat ada yang telah *built-in* di dalam PROLOG, namun beberapa predikat ada yang diimplementasikan oleh penulis. Adapun predikat-predikat yang diimplementasikan oleh penulis adalah sebagai berikut:

- `vars([Tipe], [`*`ListTypeVariable`*`])`. *Unify* [*ListTypeVariable*] dengan himpunan semua *type-variable* dari [Tipe].

```
vars(tpV(X),[tpV(X)]).
vars(tp(X,Y), Var) :-
    vars(X,Var1), vars(Y,Var2), union(Var1,Var2,Var).
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

- `varsFromList ([`*`ListType`*`], [`*`ListTypeVariable`*`])`. *Unify* [*ListTypeVariable*] dengan himpunan semua *type-variable* dari [ListType].

```
varsFromList([],[]):-!.
varsFromList([TpH | TpT], ListVar):- !,
   vars(TpH, TpHVarList),
   varsFromList(TpT, TpTVarList),
   union(TpHVarList, TpTVarList, ListVar).
```

- `freeVariable ([`*`Term`*`], [`*`ListFreeVariable`*`])`. *Unify* [*ListFreeVariable*] dengan himpunan semua *free-variable* dari [Term].

```
%FV(X) = {x}
   freeVariable(termVar(Tv),[termVar(Tv)]).
%FV ( \x.M ) = FV (M) - {x}
   freeVariable(abstraction(Head, Body),FV) :-
      freeVariable(Body, FV1), subtract(FV1,[Head],FV).
%FV(MN) = FV (M) U FV (N)
   freeVariable(application(X,Y),FV) :-
      freeVariable(X, FV1),
      freeVariable(Y, FV2),
      union(FV1,FV2,FV).
```

- `isFreeVariable ([`*`Term-Variable`*`], [`*`Term`*`])`. Predikat ini sukses jika [*Term-Variable*] merupakan *free-variable* dari [*Term*].

```
isFreeVariable(X,LambdaTerm) :-
   freeVariable(LambdaTerm,FreeVariableSet),
   member(X,FreeVariableSet).
```

- `dom ([`*`Substitution`*`],[`*`ListType`*`])`. *Unify* [*ListType*] dengan *domain* dari [*Substitution*].

```
dom([], []).
dom([s(_,Dom) | Subt_T], [ Dom|DomT]) :- dom(Subt_T,DomT).
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

- `rangeOfSubtitution ([`*Substitution*`],[`*ListType*`])`. *Unify* [*ListType*] dengan *range* dari [*Substitution*].

```
rangeOfSubtitution([],[]):- !.
rangeOfSubtitution([s(A,_)|RestUnifier], TypeVariable):-!,
   vars(A,VarsA),
   rangeOfSubtitution(RestUnifier,SubTypeVariable),
   append(VarsA, SubTypeVariable, TypeVariable).
```

- `subtitute([`*Substitution*`],[Tipe1],[Tipe2])`. *Unify* [Tipe2] dengan hasil penerapan substitusi [*Substitution*] terhadap [Tipe1].

```
%s(b) = b
   subtitute([], tpV(X), tpV(X)) :- !.
%s(alpha) = tao
   subtitute([s(A,tpV(B))|_], tpV(X), Result) :-
      X == B, !,
      Result = A.
   subtitute([s(_,tpV(B))|Subtitution], tpV(X), Result) :-
      \+ X == B, !,
      subtitute(Subtitution, tpV(X), Result).
%s(a -->b) = s(a) --> s(b).
   subtitute(Subtitution, tp(X,Y), tp(A,B)) :-
      subtitute(Subtitution, X,A),
      subtitute(Subtitution, Y,B).
```

- `subtituteSeq([`*Substitution*`],[`*ListType1*`],[`*ListType2*`])`. *Unify* [*ListType2*] dengan hasil penerapan substitusi [*Substitution*] terhadap [*ListType1*].

```
subtituteSeq(_,[],[]) :- !.
subtituteSeq(SubtitutionOP, [HT|TypeSeq], [HTN|TypeSeqRes]) :- !,
   subtitute(SubtitutionOP, HT, HTN),
   subtituteSeq(SubtitutionOP, TypeSeq, TypeSeqRes).
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

- `renameRange([`*`Substitution1`*`],[`*`Substitution2`*`],[`*`Substitution3`*`])`. *Unify* [*Substitution3*] dengan hasil substitusi penamaan (*renaming*) *range* dari [*Substitution2*] oleh [*Substitution1*].

```
renameRange(_,[],[]) :- !.
renameRange(Subtitution1,[s(A,B)|Subt2_T],
            [s(Arslt,B)|NewSubt2_T]):-
   subtitute(Subtitution1, A, Arslt),
   renameRange(Subtitution1, Subt2_T, NewSubt2_T).
```

- `composeSubtitution([`*`Substitution1`*`],[`*`Substitution2`*`],[`*`Substitution3`*`])`. *Unify* [*Substitution3*] dengan hasil komposisi substitusi dari [*Substitution1*] dan [*Substitution2*].

```
composeSubtitution(Subtitution1, Subtitution2, Result) :-
   renameRange(Subtitution1, Subtitution2, NewSubtitution2),
   removeIntersectDomain(Subtitution1, Subtitution2, NewSubt1),
   append(NewSubt1, NewSubtitution2, Result).
removeIntersectDomain(Subtitution1, Subtitution2,
                      NewSubtitution1):-
   dom(Subtitution1, DomSub1),
   dom(Subtitution2, DomSub2),
   intersection(DomSub1, DomSub2, NewDom),
   deleteSubEl(Subtitution1, NewDom, NewSubtitution1).
deleteSubEl([], _, []).
deleteSubEl([s(_,Dom) | Subt_T], ListDom, NewSubt_T) :-
   member(Dom, ListDom), !,
   deleteSubEl(Subt_T, ListDom, NewSubt_T).
deleteSubEl([s(Ran,Dom)|Subt_T],ListDom,
            [s(Ran,Dom)|NewSubt_T]):-
   \+ member(Dom, ListDom), !,
   deleteSubEl(Subt_T, ListDom, NewSubt_T).
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

- `createSubs([`*ListType*`],[`*Substitution*`])`. *Unify* [*Substitution*] dengan hasil substitusi yang *domain*-nya merupakan [*ListType*], dan *range*-nya adalah *type-variable* TPV yang dihasilkan oleh `gensym(a,TPV)`.

```
createSubs([], []) :-!.
createSubs([H|MissingVarPQ], [s(tpV(A),H)|SubOp]) :-!,
   gensym(a,A), createSubs(MissingVarPQ, SubOp).
```

- `wrapType([`*ListType*`], [`*DistintType*`], [`*CompositType*`])`. *Unify* [*CompositType*] (Sebuah tipe komposit) dengan tipe komposit yang disusun dari barisan tipe pada [*ListType*] dan [*DistintType*]. Misalkan [*ListType*] = $\langle \boldsymbol{\rho_1}, \ldots \boldsymbol{\rho_n} \rangle$ dan [*DistintType*] = $\boldsymbol{b}$, [*CompositType*] = $\boldsymbol{\rho_1} \rightarrow \ldots \rightarrow \boldsymbol{\rho_n} \rightarrow \boldsymbol{b}$

```
wrapType([],DistincType,DistincType):-!.
wrapType([H_SeqType|T_SeqType], DistincType,
   tp(H_SeqType,NewType)) :- !,
   wrapType(T_SeqType, DistincType,NewType).
```

- `extractTPVfromDedCase4([`*ListTermVar*`], [`*ListDeduction*`], [`*ListType*`])`. *Unify* [*ListType*] dengan kumpulan tipe pada setiap *type-context* dari deduksi [*ListDeduction*] yang subjeknya adalah anggota dari [*ListTermVar*].

```
%extract the type on a sequence of deduction of a lambda term,
%given a  term variable
  extractTPVfromDedCase4(_, [], []):- !.
  extractTPVfromDedCase4(IntersectionFVPQ,
                          [ded(Context,_)|_],TPVList):- !,
      extractTPVfromDed(IntersectionFVPQ, Context, TPVList).
%extract the type variable on a deduction of a lambda term,
%given a  term variable
  extractTPVfromDed( _, [], []):- !.
  extractTPVfromDed(IntersectionFVPQ,
                    [ta(TermVariable,Type)|Context],
                    [Type|TPVList]):-
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
    member(TermVariable, IntersectionFVPQ), !,
    extractTPVfromDed(IntersectionFVPQ,  Context, TPVList).
  extractTPVfromDed(IntersectionFVPQ,
                   [ta(TermVariable,_)|Context],TPVList) :-
    \+ member(TermVariable, IntersectionFVPQ), !,
    extractTPVfromDed(IntersectionFVPQ, Context, TPVList).
```

- renamingSequenceDeduction([*ListDeduction1*],[*Subt*],[*ListDeduction2*]). *Unify* [*ListDeduction2*] dengan hasil substitusi tipe oleh [*Subt*] terhadap [*ListDeduction1*].

```
%predicate for renaming a sequence of deduction
  renamingSequenceDeduction([], _, []).
  renamingSequenceDeduction([HDed | TDed], SubtitutionOp,
                            [HnewDed | TnewDed]):-
    renamingDeduction(HDed, SubtitutionOp, HnewDed),
    renamingSequenceDeduction(TDed, SubtitutionOp, TnewDed).
%predicate for renaming a deduction
  renamingDeduction(ded(Context, TA), SubtitutionOp,
                   ded( NewContext, NewTA )) :-
    renamingContext(Context, SubtitutionOp, NewContext),
    renamingTA( TA, SubtitutionOp, NewTA).
%predicate for renaming a context
  renamingContext([], _, []) :- !.
  renamingContext([Context_H | Context_T], SubtitutionOp,
                 [NewContext_H | NewContext_T]) :- !,
    renamingTA(Context_H, SubtitutionOp, NewContext_H),
    renamingContext(Context_T, SubtitutionOp, NewContext_T).
%predicate for renaming a type assignment
  renamingTA(ta(LambdaTerm, TYPE), SubtitutionOp,
            ta(LambdaTerm,NewTYPE)):-
    subtitute(SubtitutionOp, TYPE, NewTYPE).
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

- `renamingDeductionTree([`*`DeductionTree1`*`],[`*`Subt`*`],[`*`DeductionTree2`*`])`. *Unify* [*DeductionTree2*] dengan hasil substitusi tipe oleh [*Subt*] terhadap [*DeductionTree1*].

```
renamingDeductionTree(termVarDedTree(Ded_M), SubtitutionOp,
                      termVarDedTree(NewDed_M)) :-
   renamingDeduction(Ded_M, SubtitutionOp, NewDed_M).
renamingDeductionTree( abstractionDedTree(Ded_LP, BodyDedTree),
                       SubtitutionOp,
                       abstractionDedTree(NewDed_LP,
                                          NewBodyDedTree)) :-
   renamingDeduction(Ded_LP, SubtitutionOp, NewDed_LP),
   renamingDeductionTree(BodyDedTree, SubtitutionOp,
                                      NewBodyDedTree).
renamingDeductionTree( applicationDedTree(Ded_PQ, PDedVarTree,
                                          QDedVarTree),
                       SubtitutionOp,
                       applicationDedTree(NewDed_PQ, NewPDedVarTree,
                                          NewQDedVarTree)) :-
   renamingDeduction(Ded_PQ, SubtitutionOp, NewDed_PQ),
   renamingDeductionTree(PDedVarTree, SubtitutionOp, NewPDedVarTree),
   renamingDeductionTree(QDedVarTree, SubtitutionOp, NewQDedVarTree).
```

- `renamingDedVarCollection([`*`ListTypeVar1`*`],[`*`ListTypeVar2`*`],[`*`ListTypeVar3`*`], [`*`Substitution`*`])`. *Unify* [*ListTypeVar3*] dengan hasil substitusi penggantian nama (*renaming*) pada [*ListTypeVar1*] untuk setiap *type-variable* pada [*ListTypeVar1*] yang juga menjadi anggota [*ListTypeVar2*]. Kemudian predikat ini juga meng-*unify* [*Substitution*] dengan sebuah substitusi yang digunakan untuk setiap penggantian nama pada [*ListTypeVar1*].

```
renamingDedVarCollection([], _, [],[]) :- !.
renamingDedVarCollection([PDedVar_H | PDedVar_T],
                         IntersectDedVar,
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
                                [PDedVar_H|NewPDedVar_T],
                                SubtitutionOp) :-
     \+ member(PDedVar_H , IntersectDedVar), !,
     renamingDedVarCollection(PDedVar_T, IntersectDedVar,
                                   NewPDedVar_T, SubtitutionOp).
  renamingDedVarCollection([PDedVar_H | PDedVar_T],
                                IntersectDedVar,
                                [tpV(NewPDedVar_H)|NewPDedVar_T],
                                NewSubtitutionOp) :-
     member(PDedVar_H , IntersectDedVar), !,
     gensym(a,NewPDedVar_H),
     renamingDedVarCollection(PDedVar_T, IntersectDedVar,
                                   NewPDedVar_T, SubtitutionOp),
     append([s(tpV(NewPDedVar_H), PDedVar_H)], SubtitutionOp,
             NewSubtitutionOp).
```

Sedangkan predikat-predikat yang sudah *built-in* dalam PROLOG dan digunakan dalam program ini adalah sebagai berikut:

- `member /2`.
- `append /3`.
- `union /3`.
- `intersection /3`.
- `subtract /3`.
- `name /2`.
- `gensym /2`.
- `reset_gensym /2`.

Penjelasan predikat-predikat yang sudah *built-in* dalam PROLOG dapat dilihat pada [Wie07].

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

6.6 Implementasi Algoritma Unifikasi (*Unification Algorithm*)

Sub-bab ini menjelaskan implementasi algoritma unifikasi yang dijelaskan pada sub-bab 5.6.3. Algoritma ini menerima input pasangan tipe $\langle \boldsymbol{\rho}, \boldsymbol{\tau} \rangle$, dan mengeluarkan *output* kalimat yang absah (benar) bahwa $\langle \boldsymbol{\rho}, \boldsymbol{\tau} \rangle$ tidak dapat di *unify*, atau *Most General Unifier* $\boldsymbol{u}$ dari $\langle \boldsymbol{\rho}, \boldsymbol{\tau} \rangle$.

Algoritma Unifikasi ini diimplementasikan dalam predikat `findUnifier /4`. Struktur predikat `findUnifier /4` tersebut adalah sebagai berikut:

```
findUnifier(_rho, _tau, UnifiableResult, UnifierResult).
```

Predikat `findUnifier /4` memiliki 4 argumen yang dijelaskan sebagai berikut:

1. `_rho` adalah tipe masukan $\boldsymbol{\rho}$ yang ingin di-*unify*.
2. `_tau` adalah tipe masukan $\boldsymbol{\tau}$ yang ingin di-*unify*.
3. `UnifiableResult` adalah hasil unifikasi, yaitu suatu pernyataan bahwa dua tipe yang diberikan *unifiable* atau tidak.
4. `UnifierResult` adalah *unifier* yang dihasilkan algoritma ini.

Berikut adalah implementasi predikat `findUnifier /4`:

```
findUnifier(_rho, _tao, UnifiableResult, UnifierResult) :-
   compare( _rho, _tao, ComparisonResult, A, Alpha),
   findUnifierProcess( _rho, _tao, ComparisonResult, [], A,
                       Alpha, UnifiableResult, UnifierResult).
```

Di dalam predikat `findUnifier /4`, terdapat pemanggilan terhadap predikat `compare /5` dan `findUnifierProcess /8`. Predikat `compare /5` merupakan implementasi dari *comparison procedure* yang dijelaskan pada sub-bab 5.6.3.1, dan implementasinya dijelaskan pada sub-bab 6.6.1. Sedangkan predikat `findUnifierProcess /8` merupakan implementasi langkah per langkah yang dilakukan oleh algoritma unifikasi ini. Predikat ini memiliki struktur sebagai berikut:

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
findUnifierProcess(_rho,_tao,ComparisonResult,CurrentUnifier,
                    A,Alpha,UnifiableResult,UnifierResult).
```

Dalam predikat `findUnifierProcess /8`, terdapat 8 argumen yang penjelasannya adalah sebagai berikut:

1. `_rho`, merupakan tipe masukan $\boldsymbol{\rho}$ yang ingin di-*unify*.
2. `_tau`, merupakan tipe masukan $\boldsymbol{\tau}$ yang ingin di-*unify*.
3. `ComparisonResult`, merupakan hasil dari *comparison procedure* pada suatu langkah ke k. Hasil di sini adalah suatu pernyataan apakah dua tipe yang menjadi input *comparison procedure* tersebut sama atau tidak.
4. `CurrentUnifier`, merupakan *unifier* pada suatu langkah ke k yang dibangun secara bertahap dalam algoritma unifikasi ini.
5. `A`, merupakan sebuah tipe a pada *disagreement pair* $\langle \boldsymbol{a}, \boldsymbol{\alpha} \rangle$ (hasil *comparison procedure* ketika tipe yang diberikan tidak sama).
6. `Alpha`, merupakan sebuah tipe $\boldsymbol{\alpha}$ pada *disagreement pair* $\langle \boldsymbol{a}, \boldsymbol{\alpha} \rangle$ (hasil *comparison procedure* ketika tipe yang diberikan tidak sama).
7. `UnifiableResult` adalah hasil unifikasi, yaitu suatu pernyataan bahwa dua tipe yang diberikan *unifiable* atau tidak.
8. `UnifierResult` adalah *unifier* yang dihasilkan algoritma ini.

Berikut adalah implementasi dari predikat `findUnifierProcess /8` tersebut:

```
findUnifierProcess
 (_rho, _tao, ComparisonResult, CurrentUnifier, A,
  Alpha, UnifiableResult, UnifierResult) :-
   ComparisonResult == 'not_equal',
   vars(Alpha, AlphaVars),
   \+ member(A,AlphaVars), %if a bukan elemen Var(alpha)
   !,
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
    composeSubtitution([s(Alpha,A)], CurrentUnifier, NewCurrentUnifier),
    subtitute(NewCurrentUnifier, _rho, _rho2),
    subtitute(NewCurrentUnifier, _tao, _tao2),
    compare( _rho2, _tao2, NewComparisonResult, A2, Alpha2),
    findUnifierProcess
       ( _rho2, _tao2, NewComparisonResult, NewCurrentUnifier,
         A2, Alpha2, UnifiableResult, UnifierResult).
findUnifierProcess
 (_, _, ComparisonResult, _, A, Alpha, UnifiableResult, []) :-
    ComparisonResult == 'not_equal',
    vars(Alpha, AlphaVars), member(A,AlphaVars), %if a elemen Var(alpha)
    !,
    UnifiableResult = 'not_unifiable'.
findUnifierProcess
 (_, _, ComparisonResult, CurrentUnifier,
  _, _, 'unifiable', UnifierResult) :-
    ComparisonResult == 'equal',
    !,
    UnifierResult = CurrentUnifier.
```

6.6.1 *Comparison Procedure*

Sub-bab ini memaparkan implementasi *comparison procedure*. Prosedur ini merupakan suatu prosedur yang digunakan sebagai bagian dalam algoritma unifikasi. *comparison procedure* ini diimplementasikan dalam predikat `compare /5` yang memiliki struktur sebagai berikut:

```
compare( TYPE1, TYPE2, ComparisonResult, A, Alpha).
```

Dalam predikat `compare /5` tersebut terdapat 5 argumen yang penjelasannya adalah sebagai berikut:

1. `TYPE1`, merupakan tipe pertama yang menjadi argumen dari *comparison procedure*.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

2. `TYPE2`, merupakan tipe kedua yang menjadi argumen dari *comparison procedure.*

3. `ComparisonResult`, merupakan hasil dari penerapan *comparison procedure*, yaitu sebuah pernyataan apakah kedua tipe yang diberikan sama atau tidak.

4. `A`, merupakan sebuah tipe a pada *disagreement pair* $\langle \boldsymbol{a}, \boldsymbol{\alpha} \rangle$ (hasil *comparison procedure* ketika tipe yang diberikan tidak sama).

5. `Alpha`, merupakan sebuah tipe $\boldsymbol{\alpha}$ pada *disagreement pair* $\langle \boldsymbol{a}, \boldsymbol{\alpha} \rangle$ (hasil *comparison procedure* ketika tipe yang diberikan tidak sama).

Berikut adalah implementasi predikat `compare /5`:

```
compare( TYPE1, TYPE2, ComparisonResult, A, Alpha) :-
    cmp( TYPE1, TYPE2, ComparisonResult, _myu, _nu),
    decideDisagreementPair(ComparisonResult,_myu,_nu,A,Alpha).
```

Dalam implementasi predikat `compare /5`, terdapat pemanggilan kepada predikat `decideDisagreementPair /5` dan `cmp /5`. Predikat `decideDisagreementPair /5` berfungsi untuk menentukan *output disagreement pair* yang dihasilkan oleh *comparison procedure.* Sedangkan predikat `cmp /5` berfungsi untuk membandingkan kesamaan dari kedua tipe yang diberikan (sama atau tidak sama), kemudian menentukan $\boldsymbol{\mu}^*$ dan $\boldsymbol{\nu}^*$ yang menjadi a atau $\boldsymbol{\alpha}$ pada *disagreement pair* $\langle \boldsymbol{a}, \boldsymbol{\alpha} \rangle$. Predikat `cmp /5` ini memiliki struktur sebagai berikut:

```
cmp(Type1, Type2, ComparisonResult, _myu, _nu).
```

Dalam predikat `cmp /5` terdapat 5 argumen yang penjelasannya adalah sebagai berikut:

1. `Type1`, merupakan tipe pertama yang akan dibandingkan.

2. `Type2`, merupakan tipe kedua yang akan dibandingkan.

3. `ComparisonResult`, hasil perbandingan kedua tipe (sama atau tidak sama).

4. `_myu`, kandidat yang akan menjadi a atau $\boldsymbol{\alpha}$ pada *disagreement pair* $\langle \boldsymbol{a}, \boldsymbol{\alpha} \rangle$.

5. `_nu`, kandidat yang akan menjadi a atau $\boldsymbol{\alpha}$ pada *disagreement pair* $\langle \boldsymbol{a}, \boldsymbol{\alpha} \rangle$.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

Berikut adalah implementasi dari predikat `cmp /5`:

```
cmp( tpV(X) , tpV(Y) , ComparisonResult, _myu, _nu) :-
   X == Y , !,
   ComparisonResult = 'equal', _myu = tpV(X), _nu = tpV(Y).
cmp( tpV(X) , tpV(Y) , ComparisonResult, _myu, _nu) :-
   \+ X == Y, !,
   ComparisonResult = 'not_equal', _myu = tpV(X), _nu = tpV(Y).
cmp( tpV(X) , tp(P,Q) , ComparisonResult, _myu, _nu) :-
   !,
   ComparisonResult = 'not_equal', _myu = tpV(X), _nu = tp(P,Q).
cmp( tp(P,Q) , tpV(Y) , ComparisonResult, _myu, _nu) :-
   !,
   ComparisonResult = 'not_equal', _myu = tp(P,Q), _nu = tpV(Y).
cmp( tp(X,Y) , tp(P,Q) , ComparisonResult, _myu, _nu) :-
   cmp(X,P,R1,_,_), cmp(Y,Q,R2,_,_),
   R1 == 'equal', R2 == 'equal', !,
   ComparisonResult = 'equal', _myu = tp(X,Y), _nu = tp(P,Q).
cmp( tp(X,_) , tp(P,_) , ComparisonResult, _myu, _nu) :-
   cmp(X,P,R1, _myu1, _nu1), R1 == 'not_equal', !,
   ComparisonResult = 'not_equal', _myu = _myu1, _nu =  _nu1.
cmp( tp(X,Y) , tp(P,Q) , ComparisonResult, _myu, _nu) :-
   cmp(X,P,R1,_,_), R1 == 'equal',
   cmp(Y,Q,R2, _myu2, _nu2), R2 == 'not_equal', !,
   ComparisonResult = 'not_equal', _myu = _myu2, _nu = _nu2.
```

Kemudian predikat `decideDisagreementPair /5` memiliki struktur sebagai berikut:

```
decideDisagreementPair(ComparisonResult,_myu,_nu,A,Alpha).
```

Dalam predikat `decideDisagreementPair /5` terdapat 5 argumen yang penjelasannya adalah sebagai berikut:

1. `ComparisonResult`, hasil perbandingan kedua tipe (sama atau tidak sama).

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

2. _myu, kandidat yang akan menjadi a atau $\boldsymbol{\alpha}$ pada *disagreement pair* $\langle \boldsymbol{a}, \boldsymbol{\alpha} \rangle$.

3. _nu, kandidat yang akan menjadi a atau $\boldsymbol{\alpha}$ pada *disagreement pair* $\langle \boldsymbol{a}, \boldsymbol{\alpha} \rangle$.

4. A, merupakan sebuah tipe a pada *disagreement pair* $\langle \boldsymbol{a}, \boldsymbol{\alpha} \rangle$ (hasil *comparison procedure* ketika tipe yang diberikan tidak sama).

5. Alpha, merupakan sebuah tipe $\boldsymbol{\alpha}$ pada *disagreement pair* $\langle \boldsymbol{a}, \boldsymbol{\alpha} \rangle$ (hasil *comparison procedure* ketika tipe yang diberikan tidak sama).

Kemudian, berikut adalah implementasi dari predikat decideDisagreementPair /5:

```
decideDisagreementPair( ComparisonResult,tpV(X), tpV(Y), A, Alpha) :-
   ComparisonResult == 'not_equal',
   name(X,[_|XT]), name(Xnum,XT),
   name(Y,[_|YT]), name(Ynum,YT),
   Xnum =< Ynum, !,
   A = tpV(X), Alpha = tpV(Y).
decideDisagreementPair( ComparisonResult,tpV(X), tpV(Y), A, Alpha) :-
   ComparisonResult == 'not_equal',
   name(X,[_|XT]), name(Xnum,XT),
   name(Y,[_|YT]), name(Ynum,YT),
   Xnum > Ynum, !,
   A = tpV(Y), Alpha = tpV(X).
decideDisagreementPair
  ( ComparisonResult,tpV(X), tp(A,B), tpV(X), tp(A,B)) :-
   ComparisonResult == 'not_equal', !.
decideDisagreementPair
  ( ComparisonResult,tp(A,B), tpV(Y), tpV(Y), tp(A,B)) :-
   ComparisonResult == 'not_equal', !.
decideDisagreementPair( ComparisonResult,_, _, null, null) :-
   ComparisonResult == 'equal', !.
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

6.7 Optimisasi *Principal Type Algorithm*

Dalam penelitian ini dilakukan implementasi algoritma PT sesuai dengan penjelasan pada sub-bab 5.8. Setelah itu dilakukan eksplorasi. Eksplorasi tersebut menghasilkan optimisasi pada algoritma PT, dan sub-bab ini menjelaskan letak optimisasi tersebut. Kemudian hasil optimisasi tersebut diimplementasikan dan hasilnya dijelaskan pada sub-bab 6.8.

Optimisasi tersebut terletak pada penggabungan kasus II dan III dari algoritma PT. Akibat penggabungan ini, sebuah langkah dalam algoritma dihilangkan, sehingga kerja algoritma lebih efisien. Adapun penggabungan ini membuat langkah II dan III menjadi seperti berikut:

Kasus II & III.

- Jika $\boldsymbol{M \equiv \lambda x.P}$.
- Terapkan algoritma ini pada $\boldsymbol{P}$. Kemudian langkah selanjutnya adalah:
  - Jika $\boldsymbol{P}$ tidak dapat diberikan tipe, maka dapat disimpulkan bahwa $\boldsymbol{M}$ tidak dapat diberikan tipe juga.
  - Jika $\boldsymbol{P}$ memiliki *principal deduction* $\boldsymbol{\nabla_P}$, maka konklusi dari $\boldsymbol{\nabla_P}$ tersebut pasti memiliki bentuk

    $$\boldsymbol{x_1 : \alpha_1, \ldots, x_t : \alpha_t \mapsto P : \beta} \; \ldots\ldots\ldots (\boldsymbol{i})$$

    (untuk suatu tipe $\boldsymbol{\alpha_1, \ldots, \alpha_t, \beta}$).

    Atau

    $$\boldsymbol{x : \alpha, x_1 : \alpha_1, \ldots, x_t : \alpha_t \mapsto P : \beta} \; \ldots\ldots\ldots (\boldsymbol{ii})$$

    (untuk suatu tipe $\boldsymbol{\alpha, \alpha_1, \ldots, \alpha_t, \beta}$).

    Atau jika dilakukan *generalisasi* maka menjadi seperti berikut:

    $$\boldsymbol{\Gamma \mapsto P : \beta}$$

    (untuk suatu *type context* $\boldsymbol{\Gamma}$).

    Lalu langkah selanjutnya adalah:

    * Jika $\boldsymbol{x \in Subject(\Gamma)}$, maka terapkan aturan $\boldsymbol{(\rightarrow I)_{vac}}$ untuk membuat deduksi:

      $$\boldsymbol{x_1 : \alpha_1, \ldots, x_t : \alpha_t \mapsto (\lambda x.P) : \alpha \rightarrow \beta}$$

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

dimana α merupakan tipe dari x.

* Jika $x \notin Subject(\Gamma)$, pilih sebuah *type-variable* yang baru, misalkan d, yang tidak ada di ∇_P, lalu terapkan aturan $(\rightarrow I)_{vac}$ untuk membuat deduksi:

$$x_1 : \alpha_1, \ldots, x_t : \alpha_t \ \mapsto \ (\lambda x.P) : d \rightarrow \beta$$

dan kita sebut deduksi ini sebagai $\nabla_{\lambda xP}$

Mengapa penggabungan ini dianggap lebih efisien? Penggabungan ini telah menghilangkan aktifitas untuk mencari $FV(P)$ dan juga pengecekan apakah $x \in FV(P)$ atau $x \notin FV(P)$, oleh karena itu terdapat pengurangan langkah algoritma yang berujung pada efisiensi kerja algoritma. Pengecekan apakah $x \in FV(P)$ atau $x \notin FV(P)$ diperlukan untuk menentukan apakah menerapkan aturan I_{main} atau I_{vac}. Namun pada penggabungan ini, penentuan untuk menerapkan aturan I_{main} atau I_{vac} dilakukan dengan melihat *type-context* dari hasil deduksi terhadap P, sambil mencari tipe dari x.

6.8 Implementasi *Principal Type Algorithm*

Sub-bab ini menjelaskan implementasi algoritma *Principal Type* (PT) yang dijelaskan pada sub-bab 5.8 yang telah dioptimisasi seperti yang dijelaskan pada sub-bab 6.7. Implementasi algoritma PT tersebut dilakukan pada predikat `pta /5`. Struktur dari predikat `pta /5` tersebut adalah sebagai berikut:

```
pta(Term, Deduction, PTAlgorithmResult, DeductionVariable,
        TreeDeduction).
```

Dalam predikat `pta /5` tersebut, terdapat 5 argumen, yaitu:

- `Term`, merupakan λ-*term* yang diberikan sebagai *input* dari algoritma PT.
- `Deduction`, merupakan deduksi TA_λ dari *term* yang diberikan.
- `PTAlgorithmResult`, merupakan hasil penerapan algoritma PT pada *term* yang diberikan (*typable* atau tidak).

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

- `DeductionVariable`, merupakan kumpulan dari seluruh *type-variable* yang digunakan selama deduksi.
- `TreeDeduction`, merupakan *tree* yang menggambarkan deduksi dari *term* yang diberikan.

Dalam implementasi algoritma PT ini, predikat `pta /5` memiliki 3 definisi. Definisi tersebut mewakili kasus I sampai IV dalam algoritma PT (kasus II dan III digabung). Berikut ini adalah code implementasi predikat `pta /5` tersebut (beberapa predikat seperti `ptaCase2and3 /9`, dan `ptaCase4 /13` dijelaskan kemudian):

- Kasus I dari algoritma PT. Menangani *term* yang merupakan *term-variable*. Predikat ini membuat deduksi untuk *term-variable* yang diberikan.

```
%-----------------------------------------------------------------
%CASE I
%-----------------------------------------------------------------
  pta(termVar(Tv), Ded_M, Result, DedVar,
                termVarDedTree(ded([ta(termVar(Tv),tpV(A))],
                                 ta(termVar(Tv),tpV(A)))))    :-
    !,
    gensym(a,A),
    Ded_M = [ded([ta(termVar(Tv),tpV(A))],ta(termVar(Tv),tpV(A)))],
    DedVar = [tpV(A)], Result = 'typable'.
%-----------------------------------------------------------------
% END OF CASE I
%-----------------------------------------------------------------
```

- Kasus II dan III dari algoritma PT. Menangani *term* yang merupakan sebuah abstraksi. Predikat ini membuat deduksi untuk abstraksi yang diberikan. Dalam kasus ini diterapkan aturan $(\rightarrow \boldsymbol{I})$.

```
%-----------------------------------------------------------------
%CASE II & III
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
%----------------------------------------------------------------
  pta(abstraction(termVar(Tv), Body), Ded_LP, Result,
                DedVar, Ded_LPTree) :-
     !,
     pta(Body, BodyDeduction, BodyResult, BodyDedVar, BodyDedTree),
     ptaCase2and3(BodyDeduction, BodyResult, BodyDedVar,
                abstraction(termVar(Tv),Body), Ded_LP,
                Result, DedVar, BodyDedTree, Ded_LPTree).
%----------------------------------------------------------------
% END OF CASE II & III
%----------------------------------------------------------------
```

- Kasus IV dari algoritma PT. Menangani *term* yang merupakan sebuah aplikasi. Predikat ini menerapkan algoritma PT pada $\boldsymbol{P}$ dan $\boldsymbol{Q}$ dari suatu aplikasi $\boldsymbol{PQ}$. Kemudian predikat ini juga membuat deduksi untuk aplikasi *term* yang diberikan.

```
%----------------------------------------------------------------
%CASE IV
%----------------------------------------------------------------
   pta(application(P,Q), Ded_PQ, Result, DedVar, Ded_PQTree) :-
      !,
      pta(P, Ded_P, PResult, PDedVar, PDedVarTree),
      pta(Q, Ded_Q, QResult, QDedVar, PDedVarTree),
      ptaCase4(P, Ded_P, PResult, PDedVar,
               Q, Ded_Q, QResult, QDedVar,
               application(P,Q), Ded_PQ, Result, DedVar,
               PDedVarTree, QDedVarTree, Ded_PQTree).
%----------------------------------------------------------------
% END OF CASE IV
%----------------------------------------------------------------
```

Berdasarkan penjelasan algoritma PT pada sub-bab 5.8, dalam kasus II, III dan IV terdapat suatu proses pengambilan keputusan berkaitan dengan keputusan apakah

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

term yang diberikan *typable* atau tidak. Oleh karena itu terjadi percabangan proses pada algoritma tersebut. Untuk kasus II dan III, percabangan proses tersebut ditangani oleh predikat `ptaCase2and3 /9`. Sedangkan untuk kasus IV, percabangan proses tersebut ditangani oleh predikat `ptaCase4 /13`. Berikut ini adalah implementasi dari ketiga predikat tersebut:

- `ptaCase2and3 /9`. Terdapat 3 definisi untuk predikat ini. Satu definisi untuk menangani kasus ketika *body* dari abstraksi tidak dapat diberikan tipe, satu definisi untuk menangani kasus ketika *body* dari abstraksi dapat diberikan tipe, yang kemudian menerapkan $\rightarrow \boldsymbol{I_{main}}$, dan satu definisi untuk menangani kasus ketika *body* dari abstraksi dapat diberikan tipe yang kemudian menerapkan $\rightarrow \boldsymbol{I_{vac}}$. Predikat ini memiliki struktur sebagai berikut:

```
ptaCase2and3(BodyDeduction, BodyResult, BodyDedVar,
             Term, Ded_LP, Result, DedVar,
             BodyDedTree, Ded_LPTree).
```

  Dalam predikat `ptaCase2and3 /9` terdapat 9 argumen yaitu:

  - `BodyDeduction`. Merupakan hasil deduksi pada *body* dari abstraksi(*term*) yang diberikan (formula $TA_{\boldsymbol{\lambda}}$).
  - `BodyResult`. Merupakan hasil deduksi pada *body* dari abstraksi(*term*) yang diberikan (*typable* atau tidak).
  - `BodyDedVar`. Merupakan kumpulan *type-variable* yang digunakan dalam proses deduksi *body* dari abstraksi(*term*) yang diberikan.
  - `Term`. *term* yang diberikan (pasti berupa sebuah abstraksi) sebagai *input*.
  - `Ded_LP`. Merupakan hasil deduksi dari *term* yang diberikan (formula $TA_{\boldsymbol{\lambda}}$).
  - `Result`. Merupakan hasil deduksi dari *term* yang diberikan (*typable* atau tidak).
  - `DedVar`. Merupakan kumpulan *type-variable* yang digunakan dalam proses deduksi dari *term* yang diberikan.
  - `BodyDedTree`, merupakan *tree* yang menggambarkan deduksi dari *body term* yang diberikan.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

– Ded_LPTree, merupakan *tree* yang menggambarkan deduksi dari *term* yang diberikan.

Berikut adalah implementasi predikat ptaCase2and3 /9 tersebut:

– Kasus ketika *body* dari abstraksi tidak dapat diberikan tipe. Dalam hal ini, algoritma berhenti.

```
ptaCase2and3(BodyDeduction,  BodyResult, BodyDedVar,
             _, Ded_LP, Result, DedVar,
             , BodyDedTree, BodyDedTree):-
   BodyResult == 'not_typable', !,
   Ded_LP = BodyDeduction,
   DedVar = BodyDedVar,
   Result = BodyResult.
```

– Kasus ketika *body* dari abstraksi dapat diberikan tipe, yang kemudian diterapkan $\longrightarrow \boldsymbol{I_{main}}$. Dalam hal ini dibuat konstruksi $\boldsymbol{\nabla_{\lambda x P}}$ seperti yang dijelaskan pada kasus II dari algoritma PT.

```
ptaCase2and3([ded(Context, ta(BodyHead, LambdaTermType))|Tail],
             BodyResult, BodyDedVar,
             abstraction(termVar(Tv), Body),
             Ded_LP, Result, DedVar,
             BodyDedTree, Ded_LPTree):-
   BodyResult == 'typable',
   member( ta(termVar(Tv), TYPE), Context), !,
   Tp = TYPE,
   subtract(Context, [ta(termVar(Tv), Tp)], ContextLP),
   NewDed_LP = ded( ContextLP,
                    ta( abstraction(termVar(Tv), Body),
                        tp(Tp,LambdaTermType)  ) ),
      Ded_LPTree = abstractionDedTree(NewDed_LP,BodyDedTree),
   append([NewDed_LP],
         [ded(Context, ta(BodyHead, LambdaTermType) )|Tail],
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
                Ded_LP),
        DedVar = BodyDedVar, Result = 'typable'.
```

– Kasus ketika *body* dari abstraksi dapat diberikan tipe, yang kemudian diterapkan $\rightarrow \boldsymbol{I_{vac}}$. Dalam hal ini dibuat konstruksi $\boldsymbol{\nabla_{\lambda x P}}$ seperti yang dijelaskan pada kasus III dari algoritma PT.

```
ptaCase2and3([ded(Context, ta(BodyHead, LambdaTermType) )|Tail],
             BodyResult, BodyDedVar,
             abstraction(termVar(Tv), Body),
             Ded_LP, Result, DedVar,
             BodyDedTree, Ded_LPTree):-
   BodyResult == 'typable',!,
   gensym(a,A),
   NewDed_LP = ded( Context,
                    ta( abstraction(termVar(Tv),Body),
                        tp(tpV(A),LambdaTermType))),
   Ded_LPTree = abstractionDedTree(NewDed_LP,BodyDedTree),
   append([NewDed_LP],
          [ded(Context, ta(BodyHead, LambdaTermType) )|Tail],
          Ded_LP),
   append( [tpV(A)], BodyDedVar, DedVar), Result = 'typable'.
```

- `ptaCase4 /13`, Terdapat 3 definisi untuk predikat ini. Satu definisi untuk menangani kasus ketika $\boldsymbol{P}$ dari suatu aplikasi $\boldsymbol{PQ}$ tidak dapat diberikan tipe, Satu definisi untuk menangani kasus ketika $\boldsymbol{Q}$ dari suatu aplikasi $\boldsymbol{PQ}$ tidak dapat diberikan tipe, dan satu definisi untuk menangani kasus ketika $\boldsymbol{P}$ dan $\boldsymbol{Q}$ dari suatu aplikasi $\boldsymbol{PQ}$ dapat diberikan tipe. Predikat ini memiliki struktur sebagai berikut:

```
ptaCase4(Ded_P, PResult, PDedVar,
         Ded_Q, QResult, QDedVar,
         Term, Ded_PQ, Result, DedVar,
         PDedVarTree, QDedVarTree, Ded_PQTree).
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

Dalam predikat `ptaCase4 /10` terdapat 10 argumen yaitu:

- `Ded_P`. Merupakan hasil deduksi pada $\boldsymbol{P}$ dari aplikasi $\boldsymbol{PQ}$ yang diberikan (formula $\text{TA}_{\boldsymbol{\lambda}}$).
- `PResult`. Merupakan hasil deduksi pada $\boldsymbol{P}$ dari aplikasi $\boldsymbol{PQ}$ yang diberikan (*typable* atau tidak).
- `PDedVar`. Merupakan kumpulan *type-variable* yang digunakan dalam proses deduksi $\boldsymbol{P}$ dari aplikasi $\boldsymbol{PQ}$ yang diberikan.
- `Ded_Q`. Merupakan hasil deduksi pada $\boldsymbol{Q}$ dari aplikasi $\boldsymbol{PQ}$ yang diberikan (formula $\text{TA}_{\boldsymbol{\lambda}}$).
- `QResult`. Merupakan hasil deduksi pada $\boldsymbol{Q}$ dari aplikasi $\boldsymbol{PQ}$ yang diberikan (*typable* atau tidak).
- `QDedVar`. Merupakan kumpulan *type-variable* yang digunakan dalam proses deduksi $\boldsymbol{Q}$ dari aplikasi $\boldsymbol{PQ}$ yang diberikan.
- `Term`. *term* yang diberikan (pasti berupa sebuah aplikasi) sebagai *input*.
- `Ded_PQ`. Merupakan hasil deduksi dari *term* (suatu aplikasi $\boldsymbol{PQ}$) yang diberikan (formula $\text{TA}_{\boldsymbol{\lambda}}$).
- `Result`. Merupakan hasil deduksi dari *term* (suatu aplikasi $\boldsymbol{PQ}$) yang diberikan (*typable* atau tidak).
- `DedVar`. Merupakan kumpulan *type-variable* yang digunakan dalam proses deduksi dari *term* (suatu aplikasi $\boldsymbol{PQ}$)yang diberikan.
- `PDedVarTree`, merupakan *tree* yang menggambarkan deduksi dari $\boldsymbol{P}$ dari *term* yang diberikan.
- `QDedVarTree`, merupakan *tree* yang menggambarkan deduksi dari $\boldsymbol{Q}$ dari *term* yang diberikan.
- `Ded_PQTree`, merupakan *tree* yang menggambarkan deduksi dari *term* yang diberikan.

Kemudian berikut adalah implementasi predikat `ptaCase4 /13` tersebut:

- Kasus ketika $\boldsymbol{P}$ dari aplikasi $\boldsymbol{PQ}$ yang diberikan tidak dapat memiliki tipe. Dalam hal ini, algoritma berhenti.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
ptaCase4(DedP, PResult, PDedVar,
         DedQ, _, QDedVar,
         _, Ded_PQ, PResult, DedVar,
         PDedVarTree, QDedVarTree, Ded_PQTree) :-
    PResult == 'not_typable', !,
    union(PDedVar,QDedVar,DedVar),
    append(DedP,DedQ,Ded_PQ),
    Ded_PQTree = applicationDedTree([], PDedVarTree,
                                    QDedVarTree).
```

– Kasus ketika $\boldsymbol{Q}$ dari aplikasi $\boldsymbol{PQ}$ yang diberikan tidak dapat memiliki tipe. Dalam hal ini, algoritma berhenti.

```
ptaCase4(DedP, _, PDedVar,
         DedQ, QResult, QDedVar,
         _, Ded_PQ, QResult, DedVar,
         PDedVarTree, QDedVarTree, Ded_PQTree) :-
    QResult == 'not_typable', !,
    union(PDedVar,QDedVar,DedVar),
    append(DedP,DedQ,Ded_PQ),
    Ded_PQTree = applicationDedTree([], PDedVarTree,
                                    QDedVarTree).
```

– Kasus ketika $\boldsymbol{P}$ dan $\boldsymbol{Q}$ dari suatu aplikasi $\boldsymbol{PQ}$ dapat diberikan tipe. Dalam implementasi predikat ini, terdapat pemanggilan terhadap predikat `processSubCase4 /12`. Penjelasan predikat `processSubCase4 /9` dijelaskan kemudian.

```
ptaCase4(Ded_P, PResult, PDedVar,
         Ded_Q, QResult, QDedVar,
         application(P,Q), Ded_PQ, Result, DedVar,
         PDedVarTree, QDedVarTree, Ded_PQTree) :-
    PResult == 'typable', QResult == 'typable', !,
    intersection(PDedVar, QDedVar, IntersectDedVar),
    renamingDedVarCollection(PDedVar, IntersectDedVar,
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
                        NewPDedVar, SubtitutionOp),
renamingSequenceDeduction(Ded_P, SubtitutionOp, NewDed_P),
renamingDeductionTree(PDedVarTree, SubtitutionOp,
                      NewPDedVarTree),
freeVariable(P, FVP),
freeVariable(Q, FVQ),
intersection(FVP, FVQ, IntersectionFVPQ),
extractTPVfromDedCase4(IntersectionFVPQ, NewDed_P, TPVList_P),
extractTPVfromDedCase4(IntersectionFVPQ, Ded_Q, TPVList_Q),
processSubCase4( NewDed_P, NewPDedVar, TPVList_P,
                 Ded_Q, QDedVar, TPVList_Q,
                 Result, Ded_PQ, DedVar,
                 NewPDedVarTree, QDedVarTree, Ded_PQTree).
```

Kemudian, berdasarkan penjelasan algoritma PT pada sub-bab 5.8, dalam kasus IV terdapat sub kasus IVa dan IVb, serta sub kasus IVa1, IVa2, IVb1, dan IVb2. Untuk implementasi sub kasus IVa dan IVb, dilakukan pada predikat `processSubCase4 /12`, sedangkan sub kasus IVa1, IVa2, IVb1, dan IVb2 diimplementasikan pada predikat `processSubSubCase4 /14`. Berikut ini adalah implementasi dua predikat tersebut:

- `processSubCase4 /9`. Predikat ini mengimplementasikan sub kasus IVa dan IVb dari algoritma PT.
  - Sub kasus IVa. Predikat ini menerapkan langkah-langkah pada sub kasus IVa seperti yang dijelaskan pada sub-bab 5.8. Kasus IVa terjadi ketika *principal type* dari $\boldsymbol{P}$ ($\boldsymbol{PT(P)}$) komposit. Predikat ini melakukan unifikasi dengan memanggil predikat `findUnifier /4` (implementasi algoritma unifikasi).

```
% SUBCASE IV a - PT(P) is composite
processSubCase4(
         [ded(Context_P, ta( LambdaTerm_P, tp(X,Y) ) )
          | TailDedP], PDedVar, TPVList_P,
         [ded(Context_Q, ta( LambdaTerm_Q, TypeQ ) )
          | TailDedQ], QDedVar, TPVList_Q,
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
        Result, Ded_PQ, DedVar,
        PDedVarTree, QDedVarTree, Ded_PQTree) :-
    append(TPVList_P, [X], TP_P),
    append(TPVList_Q, [TypeQ], TP_Q),
    gensym(a,DistinctType),
    wrapType(TP_P, tpV(DistinctType), NewTP_P),
    wrapType(TP_Q, tpV(DistinctType), NewTP_Q),
    findUnifier(NewTP_P, NewTP_Q, UnifiableResult, UnifierResult),
    processSubSubCase4( UnifiableResult, UnifierResult,
        [ded(Context_P, ta( LambdaTerm_P, tp(X,Y) ) )
         |TailDedP], PDedVar, TP_P,
        [ded(Context_Q, ta( LambdaTerm_Q, TypeQ   ) )
         |TailDedQ], QDedVar, TP_Q,
        Result, Ded_PQ, DedVar,
        PDedVarTree,QDedVarTree,Ded_PQTree).
```

– Sub kasus IVb. Predikat ini menerapkan langkah-langkah pada sub kasus IVb seperti yang dijelaskan pada sub-bab 5.8. Kasus IVb terjadi ketika *principal type* dari $\boldsymbol{P}$ atau $\boldsymbol{PT(P)}$ atomik. Predikat ini melakukan unifikasi dengan memanggil predikat `findUnifier /4` (implementasi algoritma unifikasi).

```
% SUBCASE IV b - PT(P) is atomic
processSubCase4(
        [ded(Context_P, ta( LambdaTerm_P, tpV(X) ) )
         | TailDedP], PDedVar, TPVList_P,
        [ded(Context_Q, ta( LambdaTerm_Q, TypeQ ) )
         | TailDedQ], QDedVar, TPVList_Q,
        Result, Ded_PQ, DedVar,
        PDedVarTree, QDedVarTree, Ded_PQTree) :-
    append(TPVList_P, [tpV(X)], TP_P),
    gensym(a, NewVarA),
    append(TPVList_Q, [tp(TypeQ,tpV(NewVarA))], TP_Q),
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
        gensym(a,DistinctType),
        wrapType(TP_P, tpV(DistinctType), NewTP_P),
        wrapType(TP_Q, tpV(DistinctType), NewTP_Q),
        findUnifier(NewTP_P, NewTP_Q, UnifiableResult, UnifierResult),
        processSubSubCase4( UnifiableResult, UnifierResult,
              [ded(Context_P,  ta( LambdaTerm_P, tpV(X)  ) )
               | TailDedP], PDedVar, TP_P,
              [ded(Context_Q, ta( LambdaTerm_Q, TypeQ   ) )
               | TailDedQ], QDedVar, TP_Q,
              Result, Ded_PQ, DedVar,
              PDedVarTree, QDedVarTree, Ded_PQTree).
```

- `processSubSubCase4 /14`, Predikat ini mengimplementasikan sub kasus IVa1, IVa2, IVb1 dan IVb2 dari algoritma PT.

  – Sub kasus IVa1. Predikat ini menerapkan langkah-langkah pada sub kasus IVa1 seperti yang dijelaskan pada sub-bab 5.8. Kasus IVa1 merupakan percabangan dari kasus IVa yang terjadi ketika unifikasi yang dilakukan pada kasus IVa tidak berhasil.

```
   processSubSubCase4( UnifiableResult, _,
                       Ded_P, PDedVar, _,
                       Ded_Q, QDedVar, _,
                       Result, Ded_PQ, DedVar,
                       PDedVarTree, QDedVarTree, Ded_PQTree) :-
        UnifiableResult == 'not_unifiable',
        !,
        Result = 'not_typable',
        union(PDedVar, QDedVar, DedVar),
        append(Ded_P, Ded_Q, Ded_PQ),
        Ded_PQTree = applicationDedTree([],PDedVarTree,QDedVarTree).
```

  – Sub kasus IVa2. Predikat ini menerapkan langkah-langkah pada sub kasus IVa2 seperti yang dijelaskan pada sub-bab 5.8. Kasus IVa2 merupakan per-

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

cabangan dari kasus IVa yang terjadi ketika unifikasi yang dilakukan pada kasus IVa berhasil.

```
processSubSubCase4( UnifiableResult, Unifier,
      [ded( Context_P, ta(LambdaTerm_P, tp(X,Y)))| TailDedP],
      VarDedP, VarP,
      [ded( Context_Q, ta(LambdaTerm_Q, TypeQ) ) | TailDedQ],
      VarDedQ, VarQ,
      'typable', Ded_PQ, DedVar,
      PDedVarTree, QDedVarTree, Ded_PQTree) :-
   UnifiableResult == 'unifiable',
   !,
      append(VarP, VarQ, ListVarPQ),
      varsFromList(ListVarPQ, VarPQ),
      dom(Unifier,UnifierDomain),
      subtract(VarPQ,UnifierDomain, MissingVarPQ),
      createSubs(MissingVarPQ, AddedSubsOpForRequiredDomain),
      composeSubtitution(AddedSubsOpForRequiredDomain,
                         Unifier, CorrectDomainUnifier),
      append(VarDedP, VarDedQ, VarDedPQ),
      subtract(VarDedPQ, VarPQ, MissVarDedPQ),
      rangeOfSubtitution(Unifier, UnifierRange),
      intersection(MissVarDedPQ, UnifierRange, IntrsctRange),
      createSubs(IntrsctRange, RenamerForRange),
      renameRange(RenamerForRange,
                  CorrectDomainUnifier, NewUnifier),
      renamingSequenceDeduction(
            [ded(Context_P, ta(LambdaTerm_P, tp(X,Y)) )
             | TailDedP] , NewUnifier,
            [ded(NContext_P, ta(NLambdaTerm_P, tp(NX,NY) ) )
             | NTailDedP] ),
      renamingDeductionTree(PDedVarTree, NewUnifier,
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
                              NewPDedVarTree),
        renamingSequenceDeduction(
               [ded(Context_Q, ta(LambdaTerm_Q, TypeQ))
               | TailDedQ] , NewUnifier,
               [ded(NContext_Q, ta(NLambdaTerm_Q, NTypeQ))
               | NTailDedQ] ),
        renamingDeductionTree(QDedVarTree, NewUnifier,
                              NewQDedVarTree),
        union(NContext_P, NContext_Q, NContext),
        NewDedPQ = ded(NContext,
                    ta(application(NLambdaTerm_P, NLambdaTerm_Q),
                    NY)),
        append([ded(NContext_P, ta( NLambdaTerm_P, tp(NX,NY)))
                 | NTailDedP],
                [ded(NContext_Q, ta( NLambdaTerm_Q, NTypeQ))
                 | NTailDedQ],
                SubDedPQ),
        append([NewDedPQ], SubDedPQ, Ded_PQ),
        Ded_PQTree = applicationDedTree(NewDedPQ, NewPDedVarTree,
                                        NewQDedVarTree),
        subtituteSeq(NewUnifier, VarDedP, NewVarDedP),
        subtituteSeq(NewUnifier, VarDedQ, NewVarDedQ),
        union(NewVarDedP, NewVarDedQ, UnionDedVar),
        varsSeqType(UnionDedVar, DedVar).
```

– Sub kasus IVb1. Predikat ini menerapkan langkah-langkah pada sub kasus IVb1 seperti yang dijelaskan pada sub-bab 5.8. Kasus IVb1 merupakan percabangan dari kasus IVb yang terjadi ketika unifikasi yang dilakukan pada kasus IVb tidak berhasil.

```
processSubSubCase4( UnifiableResult, _,
                    Ded_P, PDedVar, _,
                    Ded_Q, QDedVar, _,
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
                              Result, Ded_PQ, DedVar,
                              PDedVarTree, QDedVarTree, Ded_PQTree) :-
          UnifiableResult == 'not_unifiable',
          !,
          Result = 'not_typable',
          union(PDedVar, QDedVar, DedVar),
          append(Ded_P, Ded_Q, Ded_PQ),
          Ded_PQTree = applicationDedTree([],PDedVarTree,QDedVarTree).
```

– Sub kasus IVb2. Predikat ini menerapkan langkah-langkah pada sub kasus IVb2 seperti yang dijelaskan pada sub-bab 5.8. Kasus IVb2 merupakan percabangan dari kasus IVb yang terjadi ketika unifikasi yang dilakukan pada kasus IVb berhasil.

```
\begin{verbatim}
   processSubSubCase4( UnifiableResult, Unifier,
                       [ded(Context_P, ta(LambdaTerm_P, tpV(X)))
                        | TailDedP], VarDedP, VarP,
                       [ded(Context_Q, ta(LambdaTerm_Q, TypeQ))
                        | TailDedQ], VarDedQ, VarQ,
                       'typable', Ded_PQ, DedVar,
                       PDedVarTree, QDedVarTree, Ded_PQTree) :-

                       UnifiableResult == 'unifiable',
          !,
             append(VarP, VarQ, ListVarPQ),
             varsFromList(ListVarPQ, VarPQ),
             dom(Unifier,UnifierDomain),
             subtract(VarPQ,UnifierDomain, MissingVarPQ),
             createSubs(MissingVarPQ, AddedSubsOpForRequiredDomain),
             composeSubtitution(AddedSubsOpForRequiredDomain,
                                   Unifier, CorrectDomainUnifier),
             append(VarDedP, VarDedQ, VarDedPQ),
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

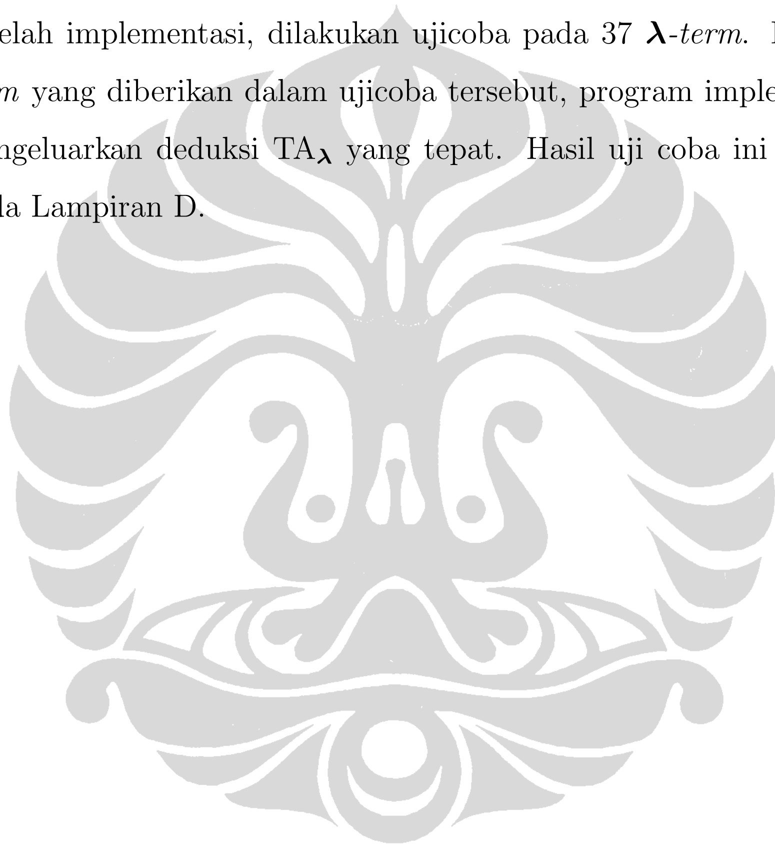
```
subtract(VarDedPQ, VarPQ, MissVarDedPQ),
rangeOfSubtitution(Unifier, UnifierRange),
intersection(MissVarDedPQ, UnifierRange, IntrsctRange),
createSubs(IntrsctRange, RenamerForRange),
renameRange(RenamerForRange,
            CorrectDomainUnifier, NewUnifier),
renamingSequenceDeduction(
   [ded(Context_P, ta(LambdaTerm_P, tpV(X)))
    | TailDedP], NewUnifier,
   [ded(NContext_P, ta( NLambdaTerm_P, Type))
    | NTailDedP] ),
renamingDeductionTree(PDedVarTree, NewUnifier,
                      NewPDedVarTree),
renamingSequenceDeduction(
   [ded(Context_Q, ta(LambdaTerm_Q, TypeQ))
    | TailDedQ], NewUnifier,
   [ded(NContext_Q, ta(NLambdaTerm_Q, NTypeQ))
    | NTailDedQ] ),
renamingDeductionTree(QDedVarTree, NewUnifier,
                      NewQDedVarTree),
union(NContext_P, NContext_Q, NContext),
      tp(_ , NewType) = Type,
NewDedPQ = ded(NContext,
            ta(application(NLambdaTerm_P, NLambdaTerm_Q),
            NewType)),
append( [ded(NContext_P, ta(NLambdaTerm_P, Type) )
          | NTailDedP],
         [ded(NContext_Q, ta(NLambdaTerm_Q, NTypeQ ) )
          | NTailDedQ],
         SubDedPQ),
append([NewDedPQ], SubDedPQ, Ded_PQ),
Ded_PQTree = applicationDedTree(NewDedPQ,NewPDedVarTree,
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

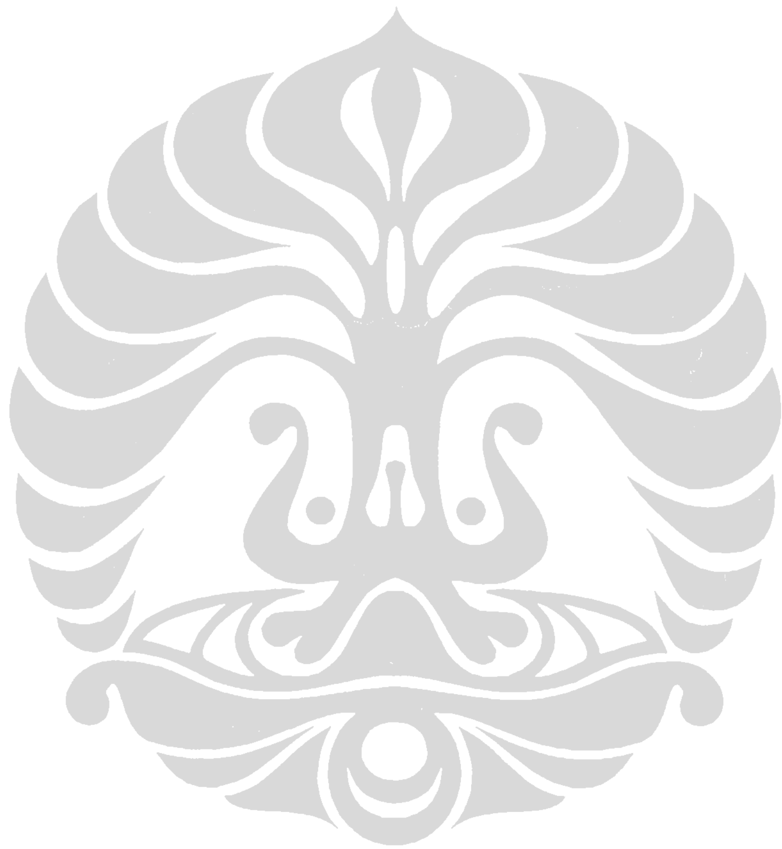
```
                                                    NewQDedVarTree),
        subtituteSeq(NewUnifier, VarDedP, NewVarDedP),
        subtituteSeq(NewUnifier, VarDedQ, NewVarDedQ),
        union(NewVarDedP, NewVarDedQ, UnionDedVar),
        varsSeqType(UnionDedVar, DedVar).
```

6.9 Uji Coba Hasil Implementasi Algoritma PT

Setelah implementasi, dilakukan ujicoba pada 37 $\boldsymbol{\lambda}$-*term*. Hasilnya, untuk setiap $\boldsymbol{\lambda}$-*term* yang diberikan dalam ujicoba tersebut, program implementasi algoritma PT ini mengeluarkan deduksi $\mathrm{TA}_{\boldsymbol{\lambda}}$ yang tepat. Hasil uji coba ini dirangkum dan disajikan pada Lampiran D.



Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008



Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

Bab 7

Typed Term

Seperti yang dijelaskan dalam bab 4, ada dua pendekatan dalam *type assignment* pada $\boldsymbol{\lambda}$-*calculus* (TA$_{\boldsymbol{\lambda}}$), yaitu pendekatan yang dilakukan oleh Curry dan Church. Pendekatan yang dilakukan oleh Church ini disebut dengan teori ***Typed-term***, dan bab ini menjelaskan secara singkat tentang pendekatan tersebut. Namun, bab ini tidak menjelaskan pendekatan Church secara keseluruhan, akan tetapi hanya memperkenalkan *typed-term* sebagai notasi alternatif untuk *TA$_{\boldsymbol{\lambda}}$-deduction*. Hal ini disebabkan karena notasi pohon yang digunakan dalam bab 4 memakan tempat yang cukup banyak (*Space-hungry deduction-tree diagram*), dan terkadang sulit untuk divisualisasikan saat deduksi yang ada sudah sangat kompleks. Walaupun notasi pohon deduksi yang digunakan dalam bab 4 dapat menggambarkan proses deduksi dengan jelas, Notasi alternatif lain yang lebih tersusun padat (rapi) juga dibutuhkan untuk memudahkan penulisan. Yang dilakukan di sini adalah hanya menggantikan notasi pohon deduksi yang memakan tempat tersebut dengan notasi *typed term* yang lebih tersusun padat dan rapi.

Untuk memenuhi kebutuhan tersebut, kita perlu mendefinisikan suatu sistem *typed-term* yang isomorfik dengan TA$_{\boldsymbol{\lambda}}$-*deduction*. Kemudian di dalam sistem tersebut berlaku:
untuk setiap $\boldsymbol{\Gamma}$, didefinisikan himpunan *typed-term* $\mathbb{TT}(\boldsymbol{\Gamma})$ yang merepresentasikan deduksi dari formula yang berbentuk:

$$\boldsymbol{\Gamma}^- \;\mapsto\; \boldsymbol{M : \tau} \qquad\qquad (\boldsymbol{\Gamma}^- \subseteq \boldsymbol{\Gamma}).$$

Acuan utama dalam bab ini adalah [Hin97], termasuk definisi-definisi, teorema dan

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

lemma yang dijelaskan pada bab ini.

7.1 Definisi *Typed Term*

Sub-bab ini menjelaskan definisi dari *typed-term.* Adapun definisinya adalah sebagai berikut:

Diberikan *type-context* Γ, himpunan $\mathbb{TT}(\Gamma)$ dari *typed-term* yang relatif terhadap Γ adalah himpunan ekspresi yang didefinisikan sebagai berikut:

1. Jika Γ mengandung $x : \sigma$, maka ekspresi x^σ merupakan anggota dari $\mathbb{TT}(\Gamma)$, dan disebut sebagai *typed variable.*

2. Jika $\Gamma_1 \cup \Gamma_2$ konsisten serta $M^{\sigma\to\tau} \in \mathbb{TT}(\Gamma_1)$ dan $N^\sigma \in \mathbb{TT}(\Gamma_2)$, maka

$$(M^{\sigma\to\tau}N^\sigma)^\tau \in \mathbb{TT}(\Gamma_1 \cup \Gamma_2)$$

3. Jika Γ konsisten dengan $\{x : \sigma\}$ dan $M^\tau \in \mathbb{TT}(\Gamma)$, maka

$$(\lambda x^\sigma.M^\tau)^{\sigma\to\tau} \in \mathbb{TT}(\Gamma - x)$$

Jika M^τ adalah *typed-term* (relatif terhadap suatu Γ), maka τ disebut sebagai tipe dari M^τ.

Contoh:

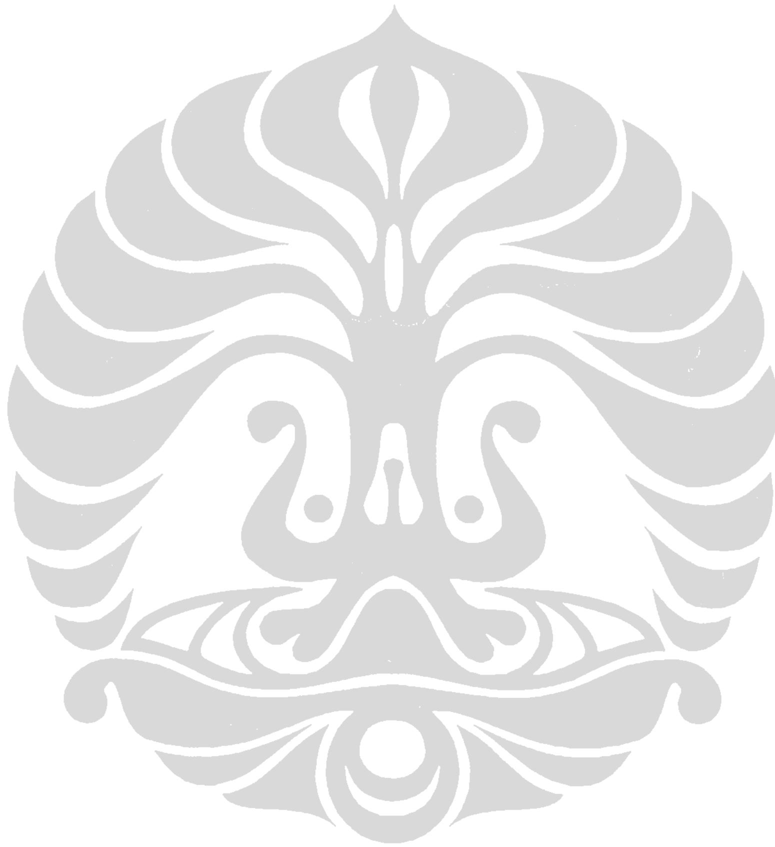
- $(\lambda x^a.x^a)^{a\to a} \in \mathbb{TT}(\emptyset)$.
- $x^a \notin \mathbb{TT}(\emptyset)$.
- $x^a \in \mathbb{TT}(\{x : a\}$.
- $(x^{(a\to a)\to b}(\lambda x^a.x^a)^{a\to a})^b \in \mathbb{TT}(\{x : (a \to a) \to b)\})$.
- $(x^{a\to b}x^a)^b$, bukanlah merupakan *typed-term.*

Kita tidak bisa selalu mengatakan bahwa $(M^{\sigma\to\tau}N^\sigma)^\tau$ juga merupakan *typed-term,* contohnya pada $(x^{d\to(a\to b)}y^d)^{a\to b}(x^{c\to a}z^c)^a$. Hal ini disebabkan karena, jika $M^{\sigma\to\tau} \in \mathbb{TT}(\Gamma_1)$ dan $N^\sigma \in \mathbb{TT}(\Gamma_2)$, kita tidak bisa menyatakan $(M^{\sigma\to\tau}N^\sigma) \in \mathbb{TT}(\Gamma_1 \cup \Gamma_2)$ kecuali $\Gamma_1 \cup \Gamma_2$ konsisten, dan pada kasus ini $\Gamma_1 \cup \Gamma_2$ tidak konsisten. Oleh karena itu $(x^{d\to(a\to b)}y^d)^{a\to b}(x^{c\to a}z^c)^a$ bukanlah merupakan *typed-term.*

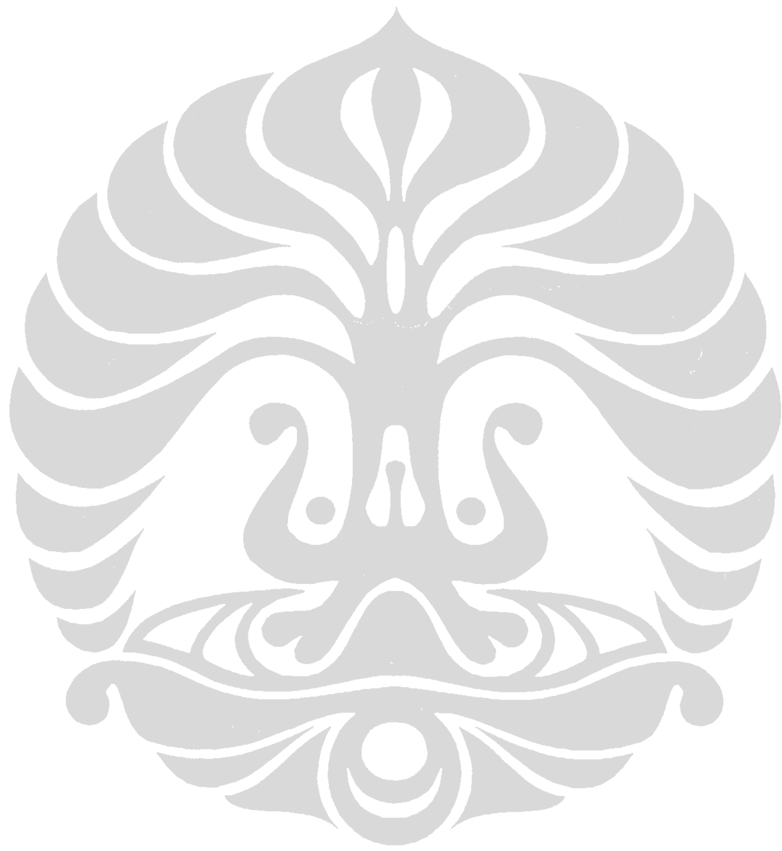
Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

7.2 Definisi *Type-Erasure* ($\boldsymbol{M}^{\not\tau}$)

Type-erasure $\boldsymbol{M}^{\not\tau}$ dari $\boldsymbol{M}^{\tau}$ adalah *term* yang tidak memiliki tipe, yang didapatkan dengan menghapus semua tipe dari $\boldsymbol{M}^{\tau}$. $\boldsymbol{M}^{\not\tau}$ merupakan subjek dari kesimpulan hasil deduksi $\mathrm{TA}_{\boldsymbol{\lambda}}$ yang direpresentasikan oleh $\boldsymbol{M}^{\tau}$.



Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008



Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

Bab 8

Algoritma Pencarian *Type Inhabitant*

Diberikan sebuah tipe τ, berapa banyak *closed term* yang dapat menerima tipe τ sebagai tipenya dalam TA_λ? Jawabannya adalah nol atau tak berhingga. Namun jika kita hanya ingin mengetahui *term* yang berada dalam *normal form* saja, maka biasanya jawabannya adalah berhingga. Ben-Yelles (1979) mengemukakan suatu algoritma yang bisa menjawab pertanyaan tersebut. Untuk setiap τ, algoritma tersebut menentukan apakah jumlah *closed term* yang berada dalam β*-normal form* yang dapat menerima tipe τ tersebut berhingga atau tidak, dan proses ini dilakukan dalam sejumlah langkah yang berhingga. Tidak hanya berhenti di situ, algoritma ini juga menghitung jumlahnya, dan mendaftarkan setiap *term* yang relevan. Algoritma ini disebut algoritma penghitungan jumlah *type inhabitant*.

Dalam algoritma penghitungan jumlah *type inhabitant*, terdapat suatu bagian yang disebut algoritma pencarian *type inhabitant*. Algoritma pencarian *type inhabitant* merupakan inti dari algoritma penghitungan jumlah *type inhabitant*, dan bab ini membahas algoritma pencarian *type inhabitant* tersebut. Penjelasan dalam bab ini diawali dengan pemaparan konsep-konsep penting yang dibutuhkan untuk memahami algoritma pencarian *inhabitant*, dan diakhiri dengan penjelasan algoritma tersebut. Acuan utama dalam bab ini adalah [Hin97], termasuk definisi-definisi, teorema dan lemma yang dijelaskan pada bab ini.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

8.1 *Inhabitant*

Sub-bab ini menjelaskan pengertian dan definisi dari *Inhabitant.* Beberapa konsep yang dipaparkan disini meliputi konsep tentang *typed* dan *untyped inhabitant*, *β-normal inhabitant*, *$\beta\eta$-normal inhabitant*, *principal inhabitant* dan juga tentang kardinalitas dari suatu himpunan.

8.1.1 *Typed* dan *Untyped Inhabitant*

Untyped inhabitant dari suatu tipe τ adalah sebuah *closed term* M sedemikian sehingga $\vdash_\lambda M : \tau$. Sedangkan ***typed inhabitant*** dari suatu tipe τ adalah *closed typed term* M^τ.

Himpunan semua *typed inhabitant* dari tipe τ disebut $Habs_t(\tau)$. Sedangkan himpunan semua *untyped inhabitant* dari tipe τ disebut $Habs_u(\tau)$.

Sebuah tipe yang memiliki setidaknya satu *inhabitant* disebut ***inhabited***.

8.1.2 *β-Normal Inhabitant*

β-normal inhabitant dari suatu tipe merupakan *inhabitant* yang berada dalam β-nf. Himpunan semua *typed normal inhabitant* dari tipe τ disebut $Nhabs_t(\tau)$. Sedangkan himpunan semua *untyped normal inhabitant* dari tipe τ disebut $Nhabs_u(\tau)$.

Antara $Habs(\tau)$ dan $Nhabs(\tau)$ berlaku:

$$Habs(\tau) \neq \emptyset \Longleftrightarrow Nhabs(\tau) \neq \emptyset.$$

8.1.3 *$\beta\eta$-Normal Inhabitant*

$\beta\eta$-normal inhabitant dari suatu tipe merupakan *inhabitant* yang berada dalam *$\beta\eta$-nf.* Himpunan semua *typed* dan *untyped $\beta\eta$-normal inhabitant* dari tipe τ disebut:

$$Nhabs_\eta(\tau).$$

8.1.4 *Lemma* tentang *Typed* dan *Untyped Inhabitant*

Jika $M^\tau \in Nhabs_t(\tau)$, maka $M^{\tau'} \in Nhabs_u(\tau)$. Lebih jauh lagi, pemetaan pada *type-erasing* merupakan korespondensi satu-satu antara *typed* dan *untyped β-normal*

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

inhabitant dari τ (*modulo* $\equiv_\alpha$), dan hal yang sama juga berlaku pada $\beta\eta$-*normal inhabitant*.

8.1.5 *Principal Inhabitant*

Sebuah *untyped inhabitant* M dari τ disebut ***principal*** jika dan hanya jika τ adalah *principal type* dari M. Sedangkan sebuah *typed inhabitant* M^τ dari τ disebut ***principal*** jika dan hanya jika hasil deduksi yang direpresentasikan oleh $\mapsto M^{\not\tau} : \tau$ adalah *principal*.

Himpunan semua *principal inhabitant* dari τ (baik yang *typed* ataupun *untyped*) disebut

$$Princ(\tau).$$

Kemudian himpunan dari semua *principal* β-*normal inhabitant* dari τ (baik yang *typed* ataupun *untyped*) disebut

$$Nprinc(\tau).$$

Lemma: M^τ disebut *typed principal* β-*normal inhabitant* dari τ jika dan hanya jika $M^{\not\tau}$ adalah *untyped principal* β-*normal inhabitant* dari τ.

Antara $Habs(\tau)$ dan $Princ\ (\tau)$ berlaku

$$Habs(\tau) \neq \emptyset \Longleftrightarrow Princ(\tau) \neq \emptyset.$$

8.1.6 Kardinalitas

Angka (0,1,2,$\ldots$ atau ∞) dari anggota dari himpunan S, atau *modulo* $\equiv_\alpha$ yang terhitung jika S adalah himpunan dari λ-*term*, disebut **kardinalitas** dari S, atau:

$$\#(S).$$

Untuk $\#(Nhabs(\tau))$ dan $\#(Nhabs_\eta(\tau))$, sering disingkat menjadi $\#(\tau)$ dan $\#_\eta(\tau)$.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

8.2 Struktur dari *typed* β-*nf*

Misalkan Γ adalah sebuah *type-context*, setiap β-*nf* $M^\tau \in \mathbb{TT}(\Gamma)$ dapat diekspresikan secara unik dalam bentuk

$$(\lambda x_1^{\tau_1} \dots x_m^{\tau_m}.(v^{(\rho_1 \to \cdots \rho_n \to \tau^*)} M_1^{\rho_1} \dots M_n^{\rho_n})^{\tau^*})^{(\tau_1 \to \cdots \tau_m \to \tau^*)},$$

dimana $m \geq 0$, $n \geq 0$, dan

$$\tau \equiv \tau_1 \to \cdots \to \tau_m \to \tau^*,$$

untuk suatu τ^* (mungkin komposit), dan setiap $M_j^{\rho_j}$ adalah sebuah β-*nf* yang memiliki tipe relatif terhadap

$$\Gamma \cup \{x_1 : \tau_1, \dots, x_m : \tau_m\}$$

(dan himpunan $\Gamma \cup \{x_1 : \tau_1, \dots, x_m : \tau_m\}$ konsisten). Lebih jauh lagi, jika M^τ merupakan *closed term*, maka $m \geq 1$ dan terdapat $i \leq m$ sedemikian sehingga

$$v \equiv x_i, \quad \tau_i \equiv \rho_1 \to \cdots \to \rho_n \to \tau^*.$$

Dalam *term*

$$(\lambda x_1^{\tau_1} \dots x_m^{\tau_m}.(v^{(\rho_1 \to \cdots \rho_n \to \tau^*)} M_1^{\rho_1} \dots M_n^{\rho_n})^{\tau^*})^{(\tau_1 \to \cdots \to \tau_m \to \tau^*)},$$

kemunculan $\lambda x_1^{\tau_1}, \dots, \lambda x_m^{\tau_m}$, $v^{(\rho_1 \to \cdots \to \rho_n \to \tau^*)}$, dan $M_1^{\rho_1}, \dots, M_n^{\rho_n}$ secara berturut-turut disebut ***initial abstractor***, ***head***, dan ***argumen***.

8.3 *Depth* dari *Typed* atau *Untyped* β-*nf*

Depth dari *typed* atau *untyped* β-*nf* didefinisikan sebagai berikut:

1. $Depth(y) = Depth(\lambda x_1 \dots x_m.y) = 0$

2. $Depth(\lambda x_1 \dots x_m.yM_1 \dots M_n) =$

   $1 + Max_{1 \leq j \leq n}(Depth(M_j))$ jika $n > 0$

Contoh:

- $Depth(\lambda uv.uxvx) = Depth(z(\lambda w.y)) = 1$.

- $Depth(\lambda x.x(z(\lambda w.y))(\lambda uv.uxvx)) = 2$.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

8.4 *Long Typed β-nf*

Sebuah *typed β-nf* M^τ disebut ***long*** atau ***maximal***, jika dan hanya jika setiap kemunculan *variable* $\underline{z}$ dalam M^τ diikuti dengan barisan terpanjang dari *argumen* yang diperbolehkan oleh tipe dari $\underline{z}$. Dengan kata lain, Sebuah *typed β-nf* M^τ disebut ***long*** atau ***maximal*** jika dan hanya jika setiap komponen dengan bentuk $(zP_1 \ldots P_n)$ $(n \geq 0)$ yang tidak berada dalam posisi fungsi memiliki tipe yang atomik. Sebuah *untyped β-nf* M disebut ***long***, relatif terhadap tipe τ, jika dan hanya jika *type-erasure* dari *term* tersebut adalah *typed long β-nf* M^τ. Himpunan dari semua *long normal inhabitant* dari τ (baik *typed* ataupun *untyped*) disebut

$$Long(\tau).$$

Contoh:

Misalkan $\tau \equiv ((a \to b) \to c) \to (a \to b) \to c$, *normal inhabitant* berikut ini tidak termasuk *long*

$$M^\tau \equiv \lambda x^{(a\to b)\to c} y^{a\to b}.x^{(a\to b)\to c} y^{a\to b}.$$

Hal ini dikarenakan $y^{a\to b}$ memiliki tipe yang membutuhkan *argumen*, tapi tidak diberikan. Namun, berikut ini adalah *term* yang berada dalam bentuk *long*:

$$M^\tau \equiv \lambda x^{(a\to b)\to c} y^{a\to b}.x^{(a\to b)\to c}(\lambda z^a.y^{a\to b} z^a).$$

Kemudian himpunan dari semua *long normal inhabitant* dari τ (baik *typed* ataupun *untyped*) dan *depth* $\leq$ d disebut

$$Long(\tau, d).$$

Lemma: Setiap *normal inhabitant* dari τ, dapat diterapkan η-*expand* menjadi *long normal inhabitant* dari τ, dan *long normal inhabitant* ini unik (*modulo* $\equiv_\alpha$). Atau dengan kata lain

$$\{M^\tau, N^\tau \in Long(\tau) \text{ dan } M^\tau =_\eta N^\tau\} \implies M^\tau \equiv_\alpha N^\tau$$

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

8.5 Ide Dasar dan Contoh dari Pencarian Inhabitant

Algoritma pencarian *inhabitant* mencari *long normal inhabitant* dari tipe τ secara bertahap dengan menaikkan *depth* (d = 0,1,2,...) pada setiap tahapnya. Strategi pencarian yang dilakukan oleh algoritma ini, didasari pada ulasan dari struktur *long typed-term* yang berada pada bentuk *β-normal form.* Pada sub-bab ini, dijelaskan ide dasar dari algoritma tersebut melalui pemaparan contoh pencarian *inhabitant*, yang didasari pada ulasan tentang struktur *long typed-term* yang berada pada bentuk *β-normal form.* Namun sebelum masuk dalam bagian tersebut, ada sebuah *lemma* dan beberapa konsep yang harus diketahui terlebih dahulu. Adapun *lemma* dan beberapa konsep tersebut adalah sebagai berikut:

Lemma: Setiap tipe τ dapat diekspresikan secara unik dalam bentuk

$$\tau \equiv \tau_1 \rightarrow \cdots \rightarrow \tau_m \rightarrow e,$$

dimana $m \geq 0$ dan e merupakan suatu atom. Kemunculan $\tau_1 \rightarrow \cdots \rightarrow \tau_m$ dan e secara berturut-turut disebut dengan ***premis*** dan ***konklusi*** atau ***tail*** dari τ, kemudian m disebut sebagai ***arity*** dari τ.

Dua kemunculan tipe disebut ***isomorphic*** jika dan hanya jika mereka merupakan kemunculan tipe yang sama. Kemudian, kita dapat mengatakan bahwa

$$Tail(\sigma) \cong Tail(\tau)$$

jika dan hanya jika komponen *tail* dari σ dan τ merupakan *isomorphic.*

Setelah memahami *lemma* dan beberapa definisi tersebut, berikut ini dipaparkan ulasan (uraian) dari struktur *long typed-term* yang berada pada bentuk *β-normal form.*

8.5.1 Ulasan (Uraian) struktur *long typed β-nf* [Hin97]

Misalkan τ adalah sembarang tipe, dan τ memiliki bentuk

$$\tau \equiv \tau_1 \rightarrow \cdots \rightarrow \tau_m \rightarrow e \qquad \text{(m} \geq 0\text{, e adalah atom).}$$

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

Kemudian misalkan M^τ merupakan sembarang *term* β-*nf* dengan tipe τ. M^τ memiliki bentuk

$$(\lambda x_1^{\tau_1} \dots x_k^{\tau_k}.(v^{(\rho_1 \to \dots \to \rho_n \to \tau^*)} M_1^{\rho_1} \dots M_n^{\rho_n})^{\tau^*})^{(\tau_1 \to \dots \to \tau_k \to \tau^*)}$$

dimana $0 \geq k \geq m$ dan $\tau^* \equiv \tau_{k+1} \to \dots \tau_m \to e$. Jika M^τ adalah *long*, maka:

1. $k = m$.
2. $\tau^* \equiv e$.
3. Tipe dari $x_1, \dots, x_m$ bertepatan dengan *premis* dari τ.
4. *Tail* dari tipe yang dimiliki v, ber-*isomorphic* dengan *tail* dari τ.
5. Jika M^τ adalah *closed-term*, maka $m \geq 1$ dan v adalah x_i $(1 \leq i \leq m)$ dan

$$\tau_i \equiv \rho_1 \to \dots \to \rho_n \to e.$$

8.5.2 Contoh pada Tipe τ dengan $\#(\tau) = 1$ [Hin97]

Tipe berikut ini:

$$\tau \equiv (a \to b \to c) \to (a \to b) \to a \to c$$

hanya memiliki tepat satu *normal inhabitant* (yang juga merupakan *long* dan *principal*), yaitu

$$S^\tau \equiv \lambda x^{a \to b \to c} y^{a \to b} z^a.xz(yz).$$

Berikut ini adalah bukti bahwa pernyataan diatas benar, kita mulai dari membuktikan bahwa $Long(\tau) = \{S^\tau\}$.

Langkah 1 Pertama kita lihat struktur dari τ, berdasarkan notasi pada 8.5.1, m = 3, dan

$$e \equiv c, \quad \tau_1 \equiv a \to b \to c, \quad \tau_2 \equiv a \to b, \quad \tau_3 \equiv a.$$

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

Oleh karena itu, $\boldsymbol{M^{\tau} \in Long(\tau)}$ harus memiliki tiga variable abstraksi, misalkan

$$\boldsymbol{M^{\tau} \equiv (\lambda x_1^{\tau_1} x_3^{\tau_3} x_3^{\tau_3} (v^{(\rho_1 \to \ldots \rho_n \to c)} M_1^{\rho_1} \ldots M_n^{\rho_n})^c)^{\tau_1 \to \tau_2 \to \tau_3 \to c}}.$$

Sekarang berdasarkan 8.5.1 (4), $\boldsymbol{v}$ haruslah salah satu dari $\boldsymbol{x_1, x_2, x_3}$ yang *tail* dari tipenya ber-*isomorphic* dengan *tail* dari $\boldsymbol{\tau}$. Dalam kasus ini, *tail* dari $\boldsymbol{\tau}$ adalah $\underline{\boldsymbol{c}}$, dan hanya $\boldsymbol{\tau_1}$ yang memiliki *tail* $\boldsymbol{c}$, oleh karena itu $\boldsymbol{v}$ haruslah $\boldsymbol{x_1}$. Selanjutnya, karena $\boldsymbol{\tau_1}$ memiliki 2 *premis*, $\boldsymbol{x_1}$ harus diikuti dengan 2 argumen. Oleh karena itu, $\boldsymbol{M}$ pasti memiliki bentuk

$$\boldsymbol{M \equiv \lambda x_1^{a \to b \to c} x_2^{a \to b} x_3^{a} . x_1^{a \to b \to c} U^a V^b}.$$

Langkah 2 Cari $\boldsymbol{U^a}$ dan $\boldsymbol{V^b}$ yang sesuai. Pertama, $\boldsymbol{U^a}$ memiliki tipe yang berbentuk atom, oleh karena itu $\boldsymbol{U^a}$ tidak mungkin merupakan suatu abstraksi, dan berdasarkan 8.5.1 (1). $\boldsymbol{U^a}$ memiliki bentuk

$$\boldsymbol{U^a \equiv (w P_1 \ldots P_r)^a} \qquad \boldsymbol{(r \geq 0)}$$

dimana $\boldsymbol{w}$ adalah $\boldsymbol{x_i}$ yang memiliki tipe yang *tail*-nya ber-*isomorphic* dengan *tail* dari tipe $\boldsymbol{U}$. *Tail* dari tipe ini merupakan $\boldsymbol{a}$, oleh karena itu satu-satunya kemungkinan adalah

$$\boldsymbol{w \equiv x_3}.$$

Kemudian, karena tipe dari $\boldsymbol{x_3}$ tidak memiliki *premis*, $\boldsymbol{r = 0}$, dan oleh karena itu

$$\boldsymbol{U^a \equiv x_3^a}.$$

Selanjutnya, kita cari $\boldsymbol{V^b}$. Karena $\boldsymbol{b}$ merupakan atom, $\boldsymbol{V^b}$ tidak mungkin sebuah abstraksi, dan berdasarkan 8.5.1 (1), $\boldsymbol{V^b}$ pasti memiliki bentuk

$$\boldsymbol{V^b \equiv (z P_1 \ldots P_r)^b} \qquad \boldsymbol{(r \geq 0)}$$

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

dimana z adalah x_i yang memiliki tipe yang *tail*-nya bertipe b, oleh karena itu satu-satunya kemungkinan adalah x_2, dan tipe dari x_2 hanya memperbolehkan 1 argumen, sehingga kita dapat

$$V^b \equiv x_2^{a\to b} W^a$$

untuk suatu W^a.

Langkah 3 Selanjutnya kita mencari W^a. Sama seperti U^a, satu-satunya kemungkinan adalah

$$W^a \equiv x_3^a$$

Kesimpulan Terdapat paling banyak satu *term* dalam $Long(\tau)$, yaitu:

$$S^\tau \equiv \lambda x_1^{a\to b\to c} x_2^{a\to b} x_3^a . x_1 x_3 (x_2 x_3).$$

(dapat terlihat bahwa *term* tersebut berada dalam $Long(\tau)$ berdasarkan definisi *Long B-nf*). Kemudian karena s^τ η*-irreducible*,

$$Nhabs(\tau) = \{S^\tau\}.$$

Lalu, berdasarkan algoritma PT, τ adalah *principal-type* dari S^τ. Oleh karena itu

$$Nprinc(\tau) = \{S^\tau\}.$$

8.5.3 Contoh pada Tipe τ dengan $\#(\tau) = $ m [Hin97]

Untuk setiap $m \geq 2$, tipe

$$\tau \equiv a \to \cdots \to a \to a \qquad (m + 1a)$$

memiliki m *normal inhabitant* yang semuanya *long* dan *non-principal*. *Normal inhabitant* dari tipe tersebut adalah

$$\textstyle\prod_i^m \equiv \lambda x_1^a \ldots x_m^a . x_i^a \qquad (1 \leq i \leq m).$$

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

Term ini disebut ***projector*** atau ***selector***. Berikut ini adalah penjelasan dari pernyataan tersebut.

Setiap $\boldsymbol{m^\tau} \in \boldsymbol{Long(\tau)}$ pasti memiliki bentuk

$$\boldsymbol{M^\tau \equiv \lambda x_1^a \ldots x_m^a . v V_1 \ldots V_n}$$

dengan $\boldsymbol{v \equiv x_i}$ untuk suatu $\boldsymbol{i \leq m}$. Namun, tipe dari $\boldsymbol{x_1^a \ldots x_m^a}$ tidak memiliki premis, oleh karena itu $\boldsymbol{n = 0}$ dan

$$\boldsymbol{M^\tau \equiv \lambda x_1^a \ldots x_m^a . x_i^a}.$$

Kemudian dapat terlihat bahwa *term* tersebut berada dalam $\boldsymbol{Long(\tau)}$, dan juga $\boldsymbol{\eta}$-*irreducible*. Oleh karena itu

$$\boldsymbol{Nhabs(\tau) = Long(\tau)}.$$

Kemudian $\boldsymbol{m \geq 2}$, dan berdasarkan algoritma PT, tidak ada *term* tersebut yang berada dalam $\boldsymbol{Nprinc(\tau)}$. Oleh karena itu

$$\boldsymbol{Nprinc(\tau) = \emptyset}.$$

8.5.4 Contoh pada Tipe $\boldsymbol{\tau}$ dengan $\boldsymbol{\#(\tau) = 0}$ [Hin97]

Tipe atomik tidak memiliki *inhabitant*. Penjelasannya adalah sebagai berikut:

Setiap tipe yang memiliki *inhabitant* pasti memiliki *normal inhabitant*. Namun, setiap tipe yang memiliki *normal inhabitant* pasti memiliki *arity* $\boldsymbol{m \geq 1}$. Oleh karena itu tidak mungkin ada tipe atomik yang memiliki *inhabitant*.

Semua tipe *skeletal* (tipe yang setiap *atom* atau *type-variable*-nya hanya muncul satu kali) tidak memiliki *inhabitant*. Penjelasannya adalah sebagai berikut:

Misalkan $\boldsymbol{\tau}$ adalah *skeletal*, dan $\boldsymbol{\tau \equiv \tau_1 \to \cdots \to \tau_m \to e \;\; (m \geq 1)}$. Jika $\boldsymbol{\tau}$ memiliki *inhabitant*, maka $\boldsymbol{\tau}$ paling tidak memiliki satu *normal inhabitant*, dan bentuknya adalah sebagai berikut

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

$$\lambda x_1 \ldots x_m.x_i M_1 \ldots M_n$$

dengan x_i memiliki tipe τ_i dan *tail* dari τ_i merupakan e. Tapi τ *skeletal*, oleh karena itu e tidak dapat muncul dalam beberapa τ_i. Jadi τ tidak memiliki *inhabitant.*

8.5.5 Contoh pada Tipe ((a→ b)→ a)→ a (Pierce *law*) [Hin97]

Tipe $\tau \equiv ((a \to b) \to a) \to a$ tidak memiliki *inhabitant.* Berikut ini adalah penjelasan dari pernyataan tersebut.

Misalkan $M^\tau \in Long(\tau)$. M^τ pasti memiliki bentuk

$$M^\tau \equiv \lambda x^{(a\to b)\to a}.vU_1 \ldots U_n \qquad (n \geq 0),$$

dan $v \equiv x$. Hal ini dikarenakan M^τ merupakan *closed-term.* Kemudian, karena tipe dari x hanya memiliki satu premis, $n = 1$. Oleh karena itu, didapatkan

$$M^\tau \equiv \lambda x^{(a\to b)\to a}.x^{(a\to b)\to a} U^{a\to b},$$

untuk suatu $U^{a\to b}$. Lalu, karena $a \to b$ hanya memiliki satu premis, $U^{a\to b}$ pasti memiliki bentuk

$$U^{a\to b} \equiv \lambda y^a.(wV_1 \ldots V_r)^b \qquad (r \geq 0).$$

Karena M^τ merupakan *closed-term*, w haruslah antara x atau y. Tapi w harus memiliki tipe yang memiliki *tail* b, dan ternyata baik x maupun y, tidak ada yang memiliki tipe tersebut, oleh karena itu tidak ada pengganti yang cocok untuk $U^{a\to b}$. Jadi, $Long(\tau) = \emptyset$ dan $Nhabs(\tau) = \emptyset$. Kemudian $Habs(\tau) = \emptyset$.

8.6 *NF-scheme*

Sub-bab ini menjelaskan pengertian dan definisi dari *nf-scheme.*

8.6.1 Definisi *NF-Scheme*

Diberikan sejumlah barisan ekspresi tak berhingga yang disebut ***meta-variable***, berbeda dari lainnya dan juga *term-variable.* *Meta-variable* dinyatakan dengan notasi sebagai berikut

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

$$"\boldsymbol{V}", "\boldsymbol{V_1}", "\boldsymbol{V_2}", \ldots$$

Kemudian, ***Nf-scheme*** didefinisikan sama seperti definisi *term* (3.2.1), kecuali dalam *nf-scheme* dapat mengandung *meta-variable* atau *term-variable* atau keduanya, dan juga *nf-scheme* harus memenuhi beberapa batasan sebagai berikut:

1. Setiap *nf-scheme* adalah $\boldsymbol{\beta}$-*nf* tanpa terjadi *bound-variable clashes.*

2. *Meta-variable* tidak boleh mengikat (muncul sebagai *binding occurences*), atau dengan kata lain $\boldsymbol{\lambda V}$ tidak diperbolehkan.

3. Dalam *nf-scheme* yang komposit, *meta-variable* hanya muncul pada posisi argumen (seperti yang didefinisikan pada 3.2.5).

4. setiap *meta-variable* dalam *nf-scheme* hanya muncul 1 kali.

Berikut ini adalah contoh dari *nf-scheme*:

- $\boldsymbol{\lambda xyz.(zV_1V_2)}$.
- $\boldsymbol{V}$.

Namun, berikut ini adalah contoh yang bukan *nf-scheme*:

- $\boldsymbol{\lambda V.V}$. Karena menyalahi batasan 2 pada definisi *NF-scheme.*
- $\boldsymbol{Vxy}$. Karena menyalahi batasan 3 pada definisi *NF-scheme.*
- $\boldsymbol{\lambda x.V}$. Karena menyalahi batasan 3 pada definisi *NF-scheme.*

8.6.2 *Proper NF-Scheme* dan *Closed NF-Scheme*

Sebuah *nf-scheme* dikatakan ***proper nf-scheme*** jika dan hanya jika *nf-scheme* tersebut memiliki setidaknya satu *meta-variable.* Kemudian, ***closed*** *nf-scheme* adalah *nf-scheme* yang tidak memiliki *free variable* (namun masih mungkin mengandung *meta-variable*, contohnya adalah $\boldsymbol{\lambda x.xV}$ dan $\boldsymbol{V}$).

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

8.6.3 *Typed NF-Scheme*

Dalam pengertian berkaitan dengan *nf-scheme*, sebuah *type-context* Γ adalah himpunan dari *assignment* yang subjeknya adalah *meta-variable* atau *term-variable*, sedemikian sehingga tidak ada subjek yang menerima lebih dari satu tipe dalam Γ. Kemudian, himpunan $\mathbb{TNS}(\Gamma)$ dari semua ***typed nf-scheme*** yang relatif terhadap Γ, didefinisikan mirip dengan $\mathbb{TT}(\Gamma)$, namun memenuhi batasan yang didefinisikan untuk *nf-scheme* (definisi batasan *nf-scheme* dijelaskan pada 8.6.1).

8.6.4 *Long Typed NF-Scheme*

Sebuah *typed nf-scheme* X^τ disebut ***long*** jika dan hanya jika setiap komponen dari X^τ dengan bentuk

$$(zY_1 \dots Y_n)^\sigma \qquad\qquad (n \geq 0)$$

yang tidak berada pada posisi fungsi , memiliki tipe yang atomik.

Contoh: *nf-scheme* (i) berikut ini adalah *long*, namun *term* (ii) bukanlah dalam bentuk *long*

(i) $x^{(a\to b)\to c}V^{a\to b}$

(ii) $x^{(a\to b)\to c}z^{a\to b}$

8.7 Algoritma Pencarian *Type Inhabitant*

Sub-bab ini menjelaskan algoritma pencarian *type inhabitant*. Adapun langkah pencarian yang dilakukan algoritma diawali dengan mencari anggota dari $Long(\tau)$ dengan *depth* $d = 0$, kemudian dilanjutkan untuk $d = 1$, $d = 2$, dan seterusnya. Dalam pencariannya, algoritma ini menghasilkan barisan berhingga $\mathscr{A}(\tau, 0)$, $\mathscr{A}(\tau, 1)$, $\mathscr{A}(\tau, 2)$, $\dots$, yang merepresentasikan himpunan ekspresi yang merupakan aproksimasi dari anggota $Long(\tau, d)$. Anggota dalam $\mathscr{A}(\tau, d)$ tersebut akan menyerupai *typed β-nf* dengan *depth* $\leq d$, namun ada kemungkinan terdapat lubang didalamnya yang akan diisi oleh algoritma ini pada langkah selanjutnya. Lubang ini direpresentasikan oleh *meta-variable*, dan aproksimasi tersebut disebut *nf-scheme* . Dalam membuat

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

$\mathscr{A}(\tau, d{+}1)$, langkah yang dilakukan adalah dengan melakukan ekstensi pada $\mathscr{A}(\tau, d)$, yaitu dengan melakukan penggantian *meta-variable*.

Sebelum masuk ke dalam penjelasan algoritma pencarian *type inhabitant*, berikut ini dipaparkan terlebih dahulu suatu teorema yang memberikan pernyataan tentang apa yang dilakukan oleh algoritma tersebut. Adapun pernyataan teorema tersebut adalah sebagai berikut:

Teorema Pencarian $\boldsymbol{Long(\tau)}$**:**

Algoritma pencarian *type inhabitant* menerima sebuah tipe τ sebagai input, dan mengeluarkan barisan (berhingga atau tak berhingga) himpunan $\mathscr{A}(\tau, d)$ (d=0,1,2,...) sedemikian sehingga untuk semua $d \geq 0$, berlaku:

1. Setiap anggota dari $\mathscr{A}(\tau, d)$ adalah *closed long typed nf-scheme* dengan tipe τ, yang merupakan salah satu diantara berikut:

   (a) Sebuah *proper nf-scheme* dengan *depth* d, atau

   (b) Sebuah *term* dengan *depth* $d - 1$.

2. $\mathscr{A}(\tau, d)$ merupakan himpunan yang berhingga.

3. $Long(\tau, d) \subseteq \mathscr{A}(\tau, 0) \cup \ldots \cup \mathscr{A}(\tau, d+1)$.

4. Jika kita sebut himpunan dari semua *term* di dalam $\mathscr{A}(\tau, d)$ sebagai "$\mathscr{A}_{term}(\tau, d)$", maka

$$Long(\tau) = \bigcup_{d \geq 0} \mathscr{A}_{term}(\tau, d)$$

Secara umum algoritma ini digambarkan pada Gambar G.1, berikut ini adalah algoritma pencarian *type inhabitant* tersebut [Hin97] [Yel79]:

Algoritma Pencarian *Type Inhabitant* (Ben-Yelles 1979)

***Input*:** Sembarang tipe τ.

- Jika τ adalah sebuah atom, maka tipe tersebut tidak memiliki *inhabitant* (algoritma berhenti).
- Jika tipe tersebut komposit, maka lakukan langkah-langkah berikut.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

Langkah 0: Pilih sembarang *meta-variable* $\boldsymbol{V}$, dan definisikan

$$\mathscr{A}(\boldsymbol{\tau}, \mathbf{0}) = \{\boldsymbol{V}^{\boldsymbol{\tau}}\}$$

Langkah d+1: Asumsikan $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$ telah didefinisikan dan memenuhi teorema pencarian $\boldsymbol{Long(\tau)}$. Kemudian, lanjutkan ke langkah berikut:

SubLangkah I Jika $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d}) = \emptyset$ atau tidak ada lagi anggota dari $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$ yang memiliki *meta-variable*, maka algoritma berhenti di sini (pada kasus ini, $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d}+\mathbf{1})$ tidak terdefinisi dan hasil dari alogritma ini hanyalah barisan berhingga $\mathscr{A}(\boldsymbol{\tau}, \mathbf{0}), \ldots, \mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$).

SubLangkah II Sebaliknya, mulai konstruksi $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d}+\mathbf{1})$ dengan mendaftarkan semua *proper nf-scheme* dalam $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$, dan terapkan IIa-IIb berikut ini pada setiap anggota tersebut (Sublangkah II digambarkan pada Gambar G.2).

SubsubLangkah IIa Diberikan sembarang *proper nf-scheme* $\boldsymbol{X}^{\boldsymbol{\tau}} \in \mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$, daftarkan semua *meta-variable* dalam $\boldsymbol{X}^{\boldsymbol{\tau}}$, misalkan hasilnya adalah

$$\boldsymbol{V}_{\mathbf{1}}^{\boldsymbol{\rho}_1}, \ldots, \boldsymbol{V}_{\boldsymbol{q}}^{\boldsymbol{\rho}_q} \qquad (\boldsymbol{q} \geq \mathbf{1}),$$

lalu terapkan IIa1-IIa1 untuk setiap *meta-variable* tersebut.

Bagian IIa1 Diberikan sembarang *meta-variable* $\boldsymbol{V}^{\boldsymbol{\rho}}$ dalam $\boldsymbol{X}^{\boldsymbol{\tau}} \in \mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$, misalkan hasilnya adalah

$$\boldsymbol{\rho} \equiv \boldsymbol{\sigma}_1 \rightarrow \cdots \rightarrow \boldsymbol{\sigma}_{\boldsymbol{m}} \rightarrow \boldsymbol{a} \qquad (\boldsymbol{m} \geq \mathbf{0}).$$

Pertama, daftarkan semua $\boldsymbol{i} \leq \boldsymbol{m}$ dimana berlaku $\boldsymbol{Tail(\sigma_i)} \cong \underline{\boldsymbol{a}} \cong \boldsymbol{Tail(\rho)}$. (Jika tidak ada sub-tipe yang memenuhi kondisi tersebut, atau $\boldsymbol{m} = \mathbf{0}$, lanjutkan ke langkah IIa2). Untuk setiap i, $\boldsymbol{\sigma_i}$ pasti memiliki bentuk

$$\boldsymbol{\sigma}_{\boldsymbol{i}} \equiv \boldsymbol{\sigma}_{\boldsymbol{i},\mathbf{1}} \rightarrow \cdots \rightarrow \boldsymbol{\sigma}_{\boldsymbol{i},\boldsymbol{n}_i} \rightarrow \boldsymbol{a} \qquad (\boldsymbol{n}_{\boldsymbol{i}} \geq \mathbf{0}).$$

Selanjutnya, definisikan

$$\boldsymbol{Y}_{\boldsymbol{i}}^{\boldsymbol{\rho}} \equiv \boldsymbol{\lambda x}_{\mathbf{1}}^{\boldsymbol{\sigma}_1} \ldots \boldsymbol{x}_{\boldsymbol{m}}^{\boldsymbol{\sigma}_m}.(\boldsymbol{x}_{\boldsymbol{i}}^{\boldsymbol{\sigma}_i} \boldsymbol{V}_{\boldsymbol{i},\mathbf{1}}^{\boldsymbol{\sigma}_{i,1}} \ldots \boldsymbol{V}_{\boldsymbol{i},\boldsymbol{n}_1}^{\boldsymbol{\sigma}_{i,n_1}})^{\boldsymbol{a}},$$

dimana $\boldsymbol{x}$ dan $\boldsymbol{V}$ adalah *term-variable* dan *meta-variable* baru yang berbeda. ($\mathrm{Y}_{\boldsymbol{i}}^{\boldsymbol{\rho}}$ disebut ***suitable replacement*** untuk $\boldsymbol{V}^{\boldsymbol{\rho}}$).

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

Bagian IIa2 Daftarkan semua *abstractor* yang menutup (*covering abstractor*) kemunculan V^{ρ} dalam X^{τ} secara terurut pada kemunculannya dalam X^{τ} dari kiri ke kanan, misalkan hasilnya adalah

$$\lambda z_1^{\zeta_1}, \ldots, \lambda z_t^{\zeta_t} \qquad (t \geq 0).$$

Kemudian, daftarkan semua $j \leq t$ (jika ada) yang berlaku $Tail(\zeta_j) \cong a$. Untuk setiap j, ζ_j memiliki bentuk

$$\zeta_j \equiv \zeta_{j,1} \to \cdots \to \zeta_{j,h_j} \to a \qquad (h_j \geq 0).$$

Lalu, definisikan

$$Z_j^{\rho} \equiv \lambda x_1^{\sigma_1} \ldots x_m^{\sigma_m}.(z_j^{\zeta_j} V_{j,1}^{\zeta_{j,1}} \ldots V_{j,h_j}^{\zeta_{j,h_j}})^a$$

dimana x dan V adalah *term-variable* dan *meta-variable* baru yang berbeda. (Y_i^{ρ} disebut ***suitable replacement*** untuk V^{ρ}).

SubsubLangkah IIb Ketika IIa1-IIa2 diaplikasikan pada semua *meta-variable* dalam X^{τ}, akan dihasilkan sejumlah *suitable replacement* untuk setiap V_i dalam X^{τ}.

Jika satu atau lebih $V_1, \ldots, V_q$ tidak memiliki *suitable replacement*, maka abaikan(buang) X^{τ}, hal ini disebut ***reject***. Kemudian mulai aplikasikan *Substep* II untuk anggota berikutnya dari $\mathscr{A}(\tau, d)$. (Sebuah aksi *reject* tidak akan menghasilkan anggota baru untuk $\mathscr{A}(\tau, d)$).

Jika semua $V_1, \ldots, V_q$ dalam X^{τ} memiliki *suitable replacement*, maka X^{τ} disebut ***extendable***. Kemudian dalam kasus ini, daftarkan semua barisan *suitable replacement* yang mungkin, misalkan hasilnya adalah

$$\langle W_1^{\rho_1}, \ldots, W_q^{\rho_q} \rangle$$

dimana W_i adalah sebuah *suitable replacement* untuk V_i dengan $i = 1, \ldots, q$. Kemudian untuk setiap barisan $\langle W_1^{\rho_1}, \ldots, W_q^{\rho_q} \rangle$, buatlah sebuah *nf-scheme* baru $X^{*\tau}$ dari X^{τ}, dengan secara bersamaan melakukan penggantian V_i dengan W_i dalam X^{τ} untuk $i = 1, \ldots, q$. (Sebut setiap barisan $\langle W_1^{\rho_1}, \ldots, W_q^{\rho_q} \rangle$ sebagai ***suitable multiple replacement***, dan sebut $X^{*\tau}$ sebagai ***ekstensi*** dari X^{τ}. Jika ekstensi ini merupakan sebuah *term*, maka disebut ***sukses***).

SubLangkah III Terakhir, Jika himpunan $\mathscr{A}(\tau, d)$ mengandung setidaknya satu *nf-scheme*, maka definisikan $\mathscr{A}(\tau, d+1)$ sebagai himpunan yang ter-

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

diri dari semua ekstensi dari semua *proper nf-scheme* yang dapat *extendable* dalam $\mathscr{A}(\tau, d)$.

8.7.1 Contoh Penggunaan Algoritma Pencarian *Type Inhabitant* dan Penjelasannya

Berikut adalah contoh penerapan algoritma pencarian *type inhabitant* pada tipe $\tau \equiv c \rightarrow b \rightarrow (b \rightarrow a) \rightarrow a$:

Langkah 0 Definisikan $\mathscr{A}(\tau, 0) = \{V^\tau\}$

Langkah 1 Mulai konstruksi $\mathscr{A}(\tau, 1)$ dengan menerapkan sublangkah IIa-IIb untuk setiap *proper Nf-scheme* dalam $\mathscr{A}(\tau, 1)$. Dalam $\mathscr{A}(\tau, 1)$ hanya terdapat satu *proper Nf-scheme* yaitu V^τ. Terapkan bagian IIa1-IIa2 pada setiap *meta-variable* dalam *proper Nf-scheme*. Dalam V^τ hanya terdapat 1 *meta-variable* yaitu V^τ. Oleh karena itu:

- Terapkan bagian IIa1 pada V^τ. Berikut adalah Penjelasannya:
  - Dalam hal ini $\tau \equiv \sigma_1 \rightarrow \sigma_2 \rightarrow \sigma_3 \rightarrow a$, dengan $\sigma_1 \equiv c$, $\sigma_2 \equiv b$, $\sigma_3 \equiv (b \rightarrow a)$.
  - Kumpulkan semua σ_i dimana berlaku $Tail(\sigma_i) \cong a \cong Tail(\tau)$, dan buat *suitable replacement* untuk setiap σ_i. Hasilnya adalah hanya $\sigma_3 \equiv (b \rightarrow a)$.
  - Buat *suitable replacement* berdasarkan σ_3. Hasilnya adalah $\lambda x_1^c x_2^b x_3^{b \rightarrow a}.(x_3^{b \rightarrow a} V_2^b)$
- Terapkan bagian IIa2 pada V^τ. Tidak ada *suitable replacement* yang bisa dihasilkan.

Selanjutnya definisikan $\mathscr{A}(\tau, 1)$ sebagai himpunan semua ekstensi *proper nf-scheme* yang *extendable*. Hasilnya $\mathscr{A}(\tau, 1) = \lambda x_1^c x_2^b x_3^{b \rightarrow a}.(x_3^{b \rightarrow a} V_2^b)$. Lanjutkan ke langkah 2.

Langkah 2 Mulai konstruksi $\mathscr{A}(\tau, 2)$ dengan menerapkan sublangkah IIa-IIb untuk setiap *proper Nf-scheme* dalam $\mathscr{A}(\tau, 2)$. Dalam $\mathscr{A}(\tau, 2)$ hanya terdapat satu

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

proper Nf-scheme yaitu $\boldsymbol{\lambda x_1^c x_2^b x_3^{b\to a}.(x_3^{b\to a} V_2^b)}$. Terapkan bagian IIa1-IIa2 pada setiap *meta-variable* dalam *proper Nf-scheme*. Dalam $\boldsymbol{\lambda x_1^c x_2^b x_3^{b\to a}.(x_3^{b\to a} V_2^b)}$ hanya terdapat 1 *meta-variable* yaitu $\boldsymbol{V_2^b}$. Oleh karena itu:

- Terapkan bagian IIa1 pada $\boldsymbol{V_2^b}$. Dalam hal ini tidak ada *premis* memiliki *tail* dari tipenya yang sama dengan $\boldsymbol{b}$. Oleh karena itu, tidak ada *suitable replacement* yang bisa dihasilkan dari bagian ini, langsung lanjutkan dengan menerapkan bagian IIa2.
- Terapkan bagian IIa2 pada $\boldsymbol{V_2^b}$. Berikut adalah penjelasannya:
  - Semua *abstractor* yang meng-*cover* $\boldsymbol{V_2^b}$ adalah $\boldsymbol{\lambda x_1^c}$, $\boldsymbol{\lambda x_2^b}$, $\boldsymbol{\lambda x_3^{b\to a}}$.
  - *abstractor* yang tipenya memiliki *tail* yang sama dengan $\boldsymbol{b}$ hanyalah $\boldsymbol{\lambda x_2^b}$.
  - Buat *suitable replacement* berdasarkan $\boldsymbol{\lambda x_2^b}$. Hasilnya adalah $\boldsymbol{\lambda x_1^c x_2^b x_3^{b\to a}.(x_3^{b\to a} x_2^b)}$

Selanjutnya definisikan $\mathscr{A}\boldsymbol{(\tau, 2)}$ sebagai himpunan semua ekstensi *proper nf-scheme* yang *extendable*. Hasilnya $\mathscr{A}\boldsymbol{(\tau, 2) = \lambda x_1^c x_2^b x_3^{b\to a}.(x_3^{b\to a} x_2^b)}$. Lanjutkan ke langkah 3.

langkah 3 Tidak ada lagi *nf-scheme* dalam $\mathscr{A}\boldsymbol{(\tau, 2)}$, karena itu tidak ada $\mathscr{A}\boldsymbol{(\tau, 3)}$ dan *output* algoritma hanyalah $\mathscr{A}\boldsymbol{(\tau, 0)}$, $\mathscr{A}\boldsymbol{(\tau, 1)}$, dan $\mathscr{A}\boldsymbol{(\tau, 2)}$.

Berikut adalah himpunan-himpunan yang dihasilkan:

$\mathscr{A}(\boldsymbol{c \to b \to (b \to a) \to a}, 0) = \{ \boldsymbol{V_1^{c\to b\to (b\to a)\to a}} \}$.

$\mathscr{A}(\boldsymbol{c \to b \to (b \to a) \to a}, 1) = \{ \boldsymbol{\lambda x_1^c x_2^b x_3^{b\to a}.(x_3^{b\to a} V_2^b)} \}$.

$\mathscr{A}(\boldsymbol{c \to b \to (b \to a) \to a}, 2) = \{ \boldsymbol{\lambda x_1^c x_2^b x_3^{b\to a}.(x_3^{b\to a} x_2^b)} \}$.

8.7.2 Contoh lain Penggunaan Algoritma Pencarian *Type Inhabitant*

Dalam sub-bab ini dipaparkan beberapa contoh penggunaan algoritma pencarian *type inhabitant* untuk mencari *type inhabitant* dari suatu tipe. Berikut adalah beberapa contoh tersebut:

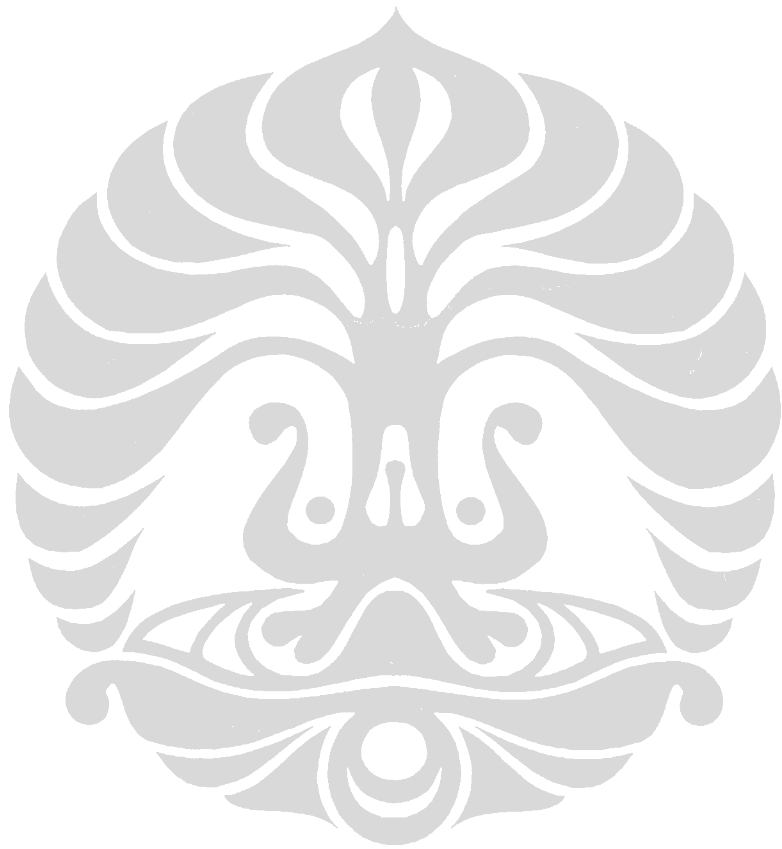
Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

- Misalkan $\tau \equiv (a \to b \to c) \to (a \to b) \to a \to c$, setelah diaplikasikan algoritma pencarian *type inhabitant*, dihasilkan himpunan-himpunan berikut ini:
  $\mathscr{A}(\tau, 0) = \{V^{\tau}\}$,
  $\mathscr{A}(\tau, 1) = \{\lambda x_1^{a \to b \to c} x_2^{a \to b} x_3^{a}.x_1^{a \to b \to c} V_1^{a} V_2^{b}\}$,
  $\mathscr{A}(\tau, 2) = \{\lambda x_1^{a \to b \to c} x_2^{a \to b} x_3^{a}.x_1^{a \to b \to c} x_3^{a} (x_2^{a \to b} V_3^{a})\}$,
  $\mathscr{A}(\tau, 3) = \{\lambda x_1^{a \to b \to c} x_2^{a \to b} x_3^{a}.x_1^{a \to b \to c} x_3^{a} (x_2^{a \to b} x_3^{a})\}$.

- Misalkan $\tau \equiv a \to \cdots \to a \to a$, setelah diaplikasikan algoritma pencarian *type inhabitant*, dihasilkan himpunan-himpunan berikut ini:
  $\mathscr{A}(\tau, 0) = \{V^{\tau}\}$,
  $\mathscr{A}(\tau, 1) = \{(\lambda x_1^{a} \ldots x_m^{a}.x_1^{a}), \ldots, (\lambda x_1^{a} \ldots x_m^{a}.x_m^{a})\}$.

- Misalkan $\tau \equiv ((a \to b) \to a) \to a$, setelah diaplikasikan algoritma pencarian *type inhabitant*, dihasilkan himpunan-himpunan berikut ini:
  $\mathscr{A}(\tau, 0) = \{V^{\tau}\}$,
  $\mathscr{A}(\tau, 1) = \{(\lambda x_1^{(a \to b) \to a}.x_1^{(a \to b) \to a} V_1^{a \to b})\}$,
  $\mathscr{A}(\tau, 2) = \emptyset$.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008



Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

Bab 9

Implementasi Algoritma Pencarian *Type Inhabitants*

Bab ini memaparkan implementasi algoritma pencarian *type inhabitants*, yaitu algoritma untuk mencari *inhabitant* dari suatu tipe yang dijelaskan pada bab 8. Penjelasan pada bab ini dimulai dengan pemaparan mengenai bentuk struktur representasi data dalam program, yang kemudian dilanjutkan dengan penjelasan alur kerja program. Setelah itu, bab ini dilanjutkan dengan penjelasan implementasi parser untuk *input* (*parser* untuk suatu tipe). Terakhir, bab ini ditutup dengan pemaparan tentang implementasi algoritma pencarian *type inhabitants* dan pengujian hasil implementasi.

9.1 Bentuk Representasi Data dalam Program

Sub-bab ini menjelaskan bagaimana data direpresentasikan dalam program. Adapun beberapa data yang perlu direpresentasikan dalam program adalah $\boldsymbol{\lambda}$-*Term* (*abstraction*, *application*, dan *term-variable*), Tipe (*composite type* dan *type-variable*), himpunan $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$, *nf-scheme*, dan *suitable replacement*.

9.1.1 Representasi $\boldsymbol{\lambda}$-*Term* dalam Program

Dalam program, $\boldsymbol{\lambda}$-*term* direpresentasikan dengan cara yang sama seperti dalam implementasi algoritma *principal type* yang dijelaskan pada bab 6. Oleh karena itu penjelasan struktur representasi $\boldsymbol{\lambda}$-*term* tidak dijelaskan ulang. Penjelasan struktur repre-

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

sentasi $\boldsymbol{\lambda}$-*term* dapat dilihat pada sub-bab 6.1.1. Namun dalam program implementasi algoritma pencarian *type inhabitant* ini, terdapat sedikit pengubahan pada struktur *term-variable.* Hal tersebut dilakukan untuk mengakomodasi kebutuhan merepresentasikan *typed-term.* Pengubahan tersebut dilakukan dengan menambahkan satu argumen dalam *functor* yang merepresentasikan *term-variable.* Pada struktur yang sebelumnya, *term-variable* direpresentasikan dengan *functor* `termVar /1`, dengan struktur sebagai berikut:

```
termVar([Nama term-variable]).
```

Namun dalam program ini, *term variable* direpresentasikan dengan *functor* `termVar /2` yang merupakan hasil modifikasi pada *functor* `termVar /1`. Modifikasi yang dilakukan adalah penambahan satu argumen yang menyatakan tipe dari *term variable* tersebut. Adapun struktur *functor* `termVar /2` tersebut menjadi:

```
termVar([Nama term-variable], [Tipe dari term-variable]).
```

9.1.2 Representasi Tipe dalam Program

Representasi sebuah tipe dalam implementasi algoritma pencarian *type inhabitant* ini sama seperti dengan struktur representasi tipe yang dijelaskan pada sub-bab 6.1.2. Oleh karena itu representasi sebuah tipe tidak dijelaskan lagi dalam sub-bab ini.

9.1.3 Representasi Himpunan $\mathscr{A}(\boldsymbol{\tau}$,d) dalam Program

$\mathscr{A}(\boldsymbol{\tau}$,d) adalah himpunan dari *nf-scheme* yang merupakan aproksimasi dari anggota $\boldsymbol{Long(\tau)}$. Di dalam implementasi algoritma pencarian *type inhabitant*, $\mathscr{A}(\boldsymbol{\tau}$,d) direpresentasikan dengan *functor* `ascript /3`. Berikut adalah struktur dari *functor* `ascript /3` tersebut:

```
ascript ([Tipe], [Depth], [List anggota 𝒜(τ,d)]),
```

dimana [*List* anggota $\mathscr{A}(\boldsymbol{\tau}$,d)] merupakan himpunan *nf-scheme* yang menjadi anggota $\mathscr{A}(\boldsymbol{\tau}$,d). Kemudian [*Depth*] merupakan sebuah bilangan bulat yang merepresentasikan nilai d dalam $\mathscr{A}(\boldsymbol{\tau}$,d). Sedangkan [Tipe] merepresentasikan sebuah tipe $\boldsymbol{\tau}$ dalam $\mathscr{A}(\boldsymbol{\tau}$,d), yang struktur representasinya dalam program dijelaskan pada sub-bab 9.1.2.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

9.1.4 Representasi *NF-scheme* dalam Program

Struktur *nf-scheme* mirip dengan **λ**-*term*, hanya saja bedanya adalah dalam *nf-scheme* terdapat *meta-variable*. Oleh karena itu, struktur data yang digunakan untuk merepresentasikan *nf-scheme* dalam implementasi algoritma pencarian *type inhabitant* sama dengan struktur data untuk merepresentasikan **λ**-*term* (seperti yang telah dijelaskan pada sub-bab 9.1.1). Namun perbedaannya adalah dalam struktur *nf-scheme* terdapat tambahan satu *functor* untuk merepresentasikan *meta-variable*. Functor tersebut adalah `metaVar /2`. Adapun struktur dari *functor* `metaVar /2` adalah sebagai berikut:

```
metaVar([nama meta-variable], [Tipe]),
```

dimana [nama *meta-variable*] merupakan nama dari *meta-variable* tersebut, dan [Tipe] merepresentasikan tipe dari *meta-variable*.

9.1.5 Representasi *Suitable Replacement* dalam Program

Suitable replacement merupakan *nf-scheme* pengganti suatu *meta-variable*. Di dalam program, *suitable replacement* direpresentasikan dengan *functor* `sr /2`, yang mana strukturnya adalah sebagai berikut:

```
sr([Meta-Variable], [List nf-scheme pengganti]),
```

dimana struktur *meta-variable* mengikuti struktur yang dijelaskan pada sub-bab 9.1.4. Sedangkan [*List nf-scheme* pengganti] merupakan kumpulan *nf-scheme* pengganti *meta-variable* yang bersangkutan.

Sebuah *meta-variable* merupakan bagian dari *nf-scheme*, dan suatu *nf-scheme* bisa memiliki lebih dari satu *meta-variable*. Oleh karena itu, untuk merepresentasikan suatu *nf-scheme* yang disertai dengan *meta-variable* dan *suitable replacement* untuk setiap *meta-variable* yang dimiliki *nf-scheme* tersebut, terdapat satu *functor* lagi. *Functor* tersebut adalah `ptmsr /2`. Adapun struktur *functor* `ptmsr /2` tersebut adalah sebagai berikut:

```
ptmsr([nf-scheme], [List suitable replacement])
```

```
ptmsr([nf-scheme], [sr([Meta-Variable], [List nf-scheme pengganti]),
    ..., sr([Meta-Variable], [List nf-scheme pengganti])])
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

dimana struktur [*nf-scheme*] sama seperti yang dijelaskan pada sub-bab 9.1.4, dan [*List suitable replacement*] merupakan *list* dari *suitable replacement* yang direpresentasikan dengan *functor* `sr /2`.

9.1.6 Representasi Pasangan *NF-Scheme* dan *Meta-Variabel*-nya dalam Program

Sebuah pasangan *nf-scheme* dan *meta-variabel* yang terkandung didalamnya direpresentasikan dengan *functor* `ptm /2`. Strukur `ptm /2` adalah sebagai berikut:

```
ptm([nf-scheme], [List metaVar])
```

9.2 Antarmuka Program

Penjelasan antarmuka pada implementasi algoritma pencarian *type inhabitant* ini sama seperti penjelasan antarmuka pada sub-bab 6.2. Hal ini dikarenakan antarmuka kedua program ini dirancang dengan kaidah-kaidah perancangan yang sama.

9.3 Alur Kerja Program

Sub-bab ini menjelaskan alur kerja program secara umum, yaitu dari saat membaca *input*, memproses *input*, dan terakhir adalah mengeluarkan (mem-*print*) *output*. Alur kerja ini digambarkan pada Gambar 9.1, dan berikut ini adalah penjelasannya:

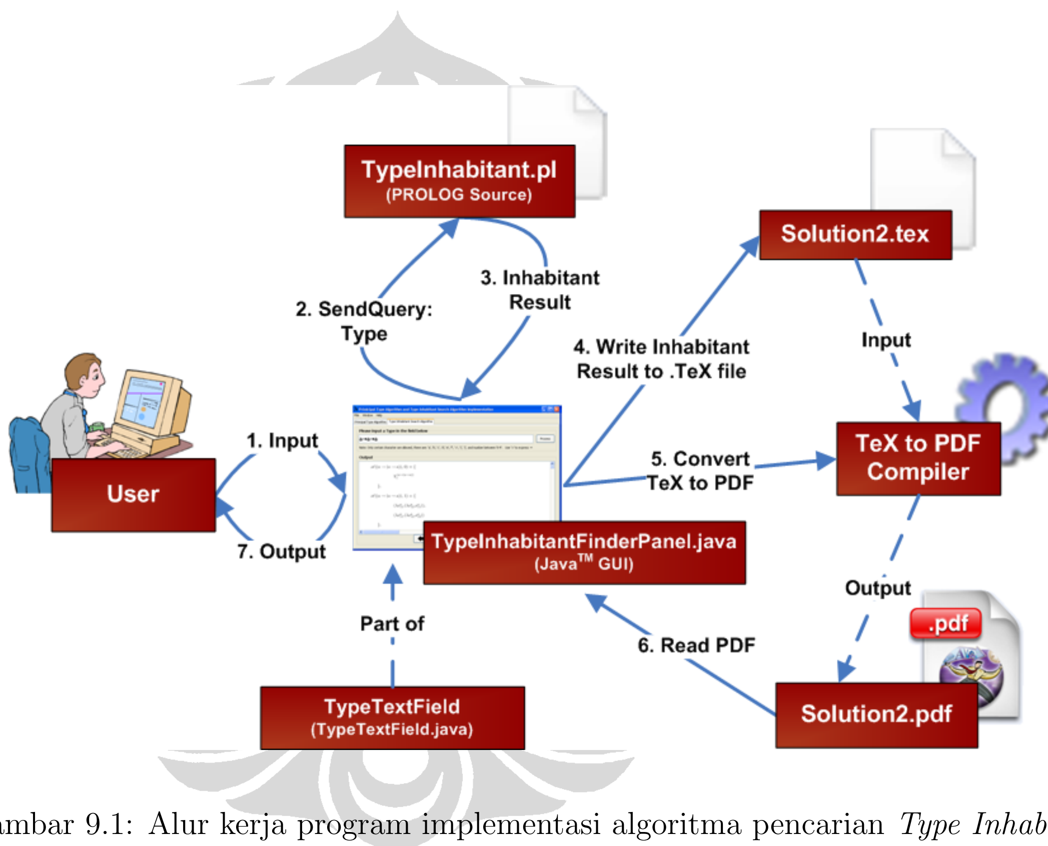
1. Penguna memberikan *input* tipe pada TypeTextField (`TypeTextField.java`).
2. `TypeInhabitantFinderPanel.java` mengirimkan *query* ke PROLOG (`TypeInhabitant.pl`) untuk mencari *inhabitant* dari tipe yang diberikan.
3. `TypeInhabitant.pl` mengirimkan *output* hasil penerapan algoritma pencarian *type inhabitant* ke `TypeInhabitantFinderPanel.java`. Output ini dalam format penulisan LaTeX.
4. `TypeInhabitantFinderPanel.java` menuliskan hasil yang diberikan `TypeInhabitant.pl` ke dalam TeX file (Solution2.tex).

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

5. `TypeInhabitantFinderPanel.java` meng-*compile* `Solution2.tex` menjadi file PDF menggunakan TeX to PDF *compiler*. Hasilnya adalah `Solution2.pdf`.

6. `TypeInhabitantFinderPanel.java` membaca `Solution2.pdf`.

7. `TypeInhabitantFinderPanel.java` menampilkan *output* hasil algoritma pencarian *type inhabitant* terhadap tipe yang diberikan kepada pengguna.



Gambar 9.1: Alur kerja program implementasi algoritma pencarian *Type Inhabitant*.

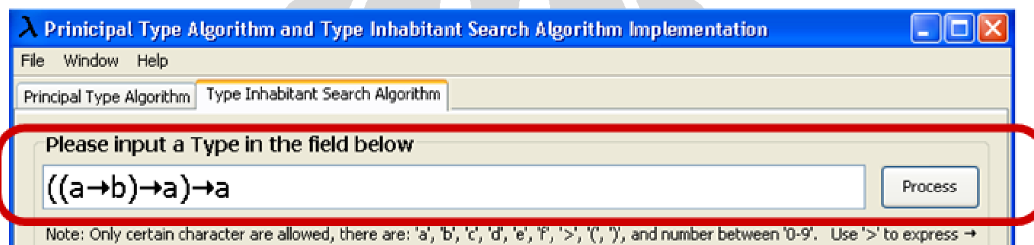
Selain alur tersebut, terdapat alur lain yang terjadi ketika algoritma pencarian *type inhabitant* telah menemukan satu *inhabitant*. Dalam kasus tersebut, program akan berhenti dan menampilkan hasil pencarian. Namun, program juga akan bertanya pada pengguna apakah pengguna akan mencari *inhabitant* lagi. Jika pengguna ingin mencari *inhabitant* berikutnya, program akan mencari *inhabitant* berikutnya, jika tidak, maka program akan berhenti.

Penjelasan lebih detail tentang implementasi untuk bagian *input*, pemrosesan dan *output* disampaikan pada sub-bab berikutnya, yaitu sub-bab 9.3.1, 9.3.2 dan 9.3.3.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

9.3.1 Bagian *Input*

Sub-bab ini menjelaskan bagian program yang menangani *input* dari pengguna. Proses ini ditangani oleh `TypeInhabitantFinderPanel.java` dan `TypeTextField.java`. `TypeTextField.java` merupakan komponen *textfield* khusus yang dibuat dengan meng-*extends* `JTextField`. `TypeTextField` merupakan *textfield* yang hanya memperbolehkan karakter tertentu untuk dimasukkan, yaitu: 'a', 'b', 'c', 'd', 'e', 'f', '>', '(', ')', '1', '2', '3', '4', '5', '6', '7', '8', '9', '0'. Selain itu, `TypeTextField` juga akan menampilkan karakter '>' sebagai $\longrightarrow$. Gambar 9.2 menunjukkan *screenshot* dari `TypeTextField` tersebut.



Gambar 9.2: Gambar *type textfield*.

9.3.2 Bagian Pemrosesan

Ketika pengguna menekan tombol proses, `TypeInhabitantFinderPanel.java` akan membuat sebuah *List* yang nantinya menjadi *input* dari *parser* untuk tipe pada `TypeInhabitant.pl`, dan kemudian mengirimkan *query* ke `TypeInhabitant.pl`. *Query* tersebut memanggil predikat `applyInhabitantSearchAlgoritm /1`. Predikat inilah yang melakukan pemrosesan *input*. Dalam predikat ini dilakukan *parsing* terhadap *input* untuk membentuk *tree* dari tipe yang diberikan, yang kemudian menjadi *input* dari algoritma pencarian *type inhabitant*. Adapun implementasi predikat tersebut adalah sebagai berikut:

```
applyInhabitantSearchAlgoritm(TypeInListFormat) :-
    reset_gensym(meta),
    type(Tree, TypeInListFormat, [] ), !,
    inhabitant(Tree,_).
applyInhabitantSearchAlgoritm(_) :-
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
write('invalid_input;').
```

Dalam `applyInhabitantSearchAlgoritm /1` terdapat satu argumen yaitu [TypeInListFormat]. [TypeInListFormat] adalah *input* tipe dalam bentuk *List*.

9.3.3 Bagian *Output*

Sub-bab ini menjelaskan bagian program yang menangani *output* hasil pemrosesan. *Output* program adalah sekumpulan himpunan $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$ (dengan d = 0, 1, 2, ...). *Output* ini dikeluarkan di dalam predikat `inhabitant /2` sembari melakukan pemrosesan. Dalam mengeluarkan *output*, digunakan predikat `generateStringForSeqAscript /2` untuk melakukan *formating* terhadap *output*. Predikat tersebut merupakan predikat yang mentranslasikan barisan $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$ yang direpresentasikan dengan struktur data seperti yang dijelaskan pada sub-bab 9.1, menjadi sebuah *string* dalam format LaTeX. Selain predikat tersebut, terdapat juga beberapa predikat lain yang memiliki fungsi yang sama. Predikat tersebut adalah:

- `generateStringForSeqAscript /2`. Mentranslasikan *list* dari himpunan $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$ yang direpresentasikan dalam bentuk *list* dari *functor* `ascript /3`, menjadi bentuk tulisan (*string*) dalam format LaTeX. Berikut adalah implementasinya:

```
generateStringForSeqAscript([], '') :-  !.
generateStringForSeqAscript([AscrH|[]], StringResult) :- !,
   generateStringForAscript(AscrH, Str),
   atom_concat(Str,'',StringResult).
generateStringForSeqAscript([AscrH|AscrT], StringResult) :- !,
   generateStringForAscript(AscrH, Str),
   generateStringForSeqAscript(AscrT,SubResult),
   atom_concat(Str,'\n\n',StrResult),
   atom_concat(StrResult, SubResult, StringResult).
```

- `generateStringForAscript /2`. Mentranslasikan himpunan $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$ yang direpresentasikan dalam *functor* `ascript /3`,menjadi bentuk tulisan (*string*) dalam format LaTeX. Berikut adalah implementasinya:

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
generateStringForAscript(ascript(Type,Depth,Member),Str) :-
   Member == [], !,
   generateStringforType(Type, StrType),
   atom_concat('$\\mathscr{A}$($', StrType, Str1),
   atom_concat(Str1, ' $, ', Str2),
   atom_concat(Str2, Depth, Str3),
   atom_concat(Str3, ') = \\{ \\}. \\\\ ', Str).
generateStringForAscript(ascript(Type,Depth,Member),Str) :-
   Member \== [], !,
   generateStringforType(Type, StrType),
   generateStringForAscriptMember(Member, AscriptMemberStr),
   atom_concat('$\\mathscr{A}$($', StrType, Str1),
   atom_concat(Str1, ' $, ', Str2),
   atom_concat(Str2, Depth, Str3),
   atom_concat(Str3, ') = \\{ \\\\ \n\n ', Str4),
   atom_concat(Str4, AscriptMemberStr, Str5),
   atom_concat(Str5, ' \n\n \\hspace*\{0.5cm\} \\}. \\\\ ', Str).
```

- `generateStringForAscriptMember /2`. Mentranslasikan himpunan anggota $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$ yang direpresentasikan dalam *list nf-scheme*, menjadi bentuk tulisan (*string*) dalam format LaTeX. Berikut adalah implementasinya:

```
generateStringForAscriptMember([],'') :- !.
generateStringForAscriptMember([AscriptMember|[]],Str) :-!,
   generateStringForAnNfScheme(AscriptMember,StrNf),
   atom_concat('\\hspace*\{2cm\}$',StrNf,A),
   atom_concat(A,'$ \\\\ ',Str).
generateStringForAscriptMember([AscrMmbr_H|AscrMmbr_T],Str) :-
   generateStringForAnNfScheme(AscrMmbr_H, StrNf),
   generateStringForAscriptMember(AscrMmbr_T, SubResult),
   atom_concat('\\hspace*\{2cm\}$',StrNf,A),
   atom_concat(A,'$, \\\\ \n\n ',B),
   atom_concat(B,SubResult,Str).
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

- `generateStringForAnNfScheme /2`. Mentranslasikan sebuah *nf-scheme* yang direpresentasikan dalam bentuk struktur yang dijelaskan pada sub-bab 9.1.4, menjadi bentuk tulisan (*string*) dalam format LaTeX. Berikut adalah implementasinya:

```
generateStringForAnNfScheme(metaVar(X,Type),Str) :- !,
   atom_concat(meta,MetNum,X),
   generateStringforType(Type, TypeStr),
   atom_concat('V', '^{', Str1),
   atom_concat(Str1, TypeStr, Str2 ),
   atom_concat(Str2, '}_{', Str3),
   atom_concat(Str3, MetNum, Str4),
   atom_concat(Str4, '}', Str).
generateStringForAnNfScheme(termVar(X,Type),Str) :- !,
   atom_concat(x,VarNum,X),
   generateStringforType(Type, TypeStr),
   atom_concat('x', '^{', Str1),
   atom_concat(Str1, TypeStr, Str2 ),
   atom_concat(Str2, '}_{', Str3),
   atom_concat(Str3, VarNum, Str4),
   atom_concat(Str4, '}', Str).
generateStringForAnNfScheme(abstraction(X,Y),StrResult) :-  !,
   generateStringForAnNfScheme(X,XS),
   generateStringForAnNfScheme(Y,YS),
   atom_concat('(',' \\lambda ',A), atom_concat(A,XS,B),
   atom_concat(B,'.',C),atom_concat(C,YS,D),
   atom_concat(D,')',StrResult).
generateStringForAnNfScheme(application(X,Y),StrResult) :-  !,
   generateStringForAnNfScheme(X,XS),
   generateStringForAnNfScheme(Y,YS),
   atom_concat('(',XS,A), atom_concat(A,YS,B),
   atom_concat(B,')',StrResult).
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

- `generateStringforType /2`. Mentranslasikan sebuah tipe m yang direpresentasikan dalam bentuk struktur yang dijelaskan pada sub-bab 9.1.2,menjadi bentuk tulisan (*string*) dalam format LaTeX. Berikut adalah implementasinya:

```
generateStringforType(tpV(X), X) :- !.
generateStringforType(tp(X,Y), StringResult) :- !,
   generateStringforType(X, XS),
   generateStringforType(Y,YS),
   atom_concat('(',XS, XSA),
   atom_concat(YS, ')', YSA),
   atom_concat(XSA,' \\rightarrow ',XSP),
   atom_concat(XSP,YSA,StringResult).
```

9.4 Implementasi *Parser* untuk Tipe

Sub-bab ini menjelaskan implementasi *parser* untuk tipe. *Parser* ini diimplementasikan menggunakan *Definite Clause Grammar*(DCG) dalam PROLOG. *Code* implementasi lengkap dari *parser* tipe ini dapat dilihat pada Lampiran C.

Parser untuk tipe ini berfungsi untuk menguraikan *input* dari pengguna, dan menentukan apakah *input* yang diberikan merupakan suatu tipe yang benar sesuai dengan pengertian tipe tersebut. Selain itu, *parser* juga membuat konstruksi *tree* yang merepresentasikan struktur dari tipe yang bersangkutan. Parser untuk tipe ini dibangun berdasarkan definisi dari tipe yang dijelaskan pada sub-bab 4.1.1. Menurut definisi tersebut, sebuah tipe dapat berupa *type-variable* atau suatu tipe komposit. Oleh karena itu, dibuatlah spesifikasi DCG sebagai berikut untuk menyatakan hal tersebut:

- Sebuah *type-variable* merupakan sebuah tipe.

```
type(X) --> typeVariable(X).
```

- Sebuah tipe komposit merupakan sebuah tipe.

```
type(X) --> compositeType(X).
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

Dalam spesifikasi di atas, setiap simbol *non-terminal* memiliki argumen X. Argumen X tersebut merupakan struktur *tree* yang merepresentasikan sebuah tipe, yang dibangun oleh *parser* ini saat menguraikan *input*. Selanjutnya, implementasi DCG untuk *parser* tipe ini dilanjutkan dengan spesifikasi untuk *type-variable* dan tipe komposit.

Berdasarkan penjelasan tentang notasi penulisan tipe yang dijelaskan pada sub-bab 4.1.2, sebuah *type-variable* dituliskan dengan huruf "a", "b", "c", "d", "e", "f", "g" dengan atau tanpa *subscript* angka. Dari penjelasan tersebut, disusunlah DCG sebagai berikut:

- Sebuah *type-variable* yang berada di dalam tanda kurung juga merupakan *type-variable*.

```
typeVariable(TV) --> ['('], typeVariable(TV), [')'].
```

- Sebuah *type-variable* direpresentasikan dengan huruf kecil a, b, c, d, e, atau f, tanpa *subscript* angka.

```
typeVariable(tpV(a)) --> [a].
typeVariable(tpV(b)) --> [b].
typeVariable(tpV(c)) --> [c].
typeVariable(tpV(d)) --> [d].
typeVariable(tpV(e)) --> [e].
typeVariable(tpV(f)) --> [f].
```

- Sebuah *term-variable* direpresentasikan dengan huruf kecil a, b, c, d, e, atau f, dengan *subscript* angka.

```
typeVariable(tpV(TypeVar)) -->
   [a], number(N), {atom_concat('a',N,TypeVar)}.
typeVariable(tpV(TypeVar)) -->
   [b], number(N), {atom_concat('b',N,TypeVar)}.
typeVariable(tpV(TypeVar)) -->
   [c], number(N), {atom_concat('c',N,TypeVar)}.
typeVariable(tpV(TypeVar)) -->
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
   [d], number(N), {atom_concat('d',N,TypeVar)}.
typeVariable(tpV(TypeVar)) -->
   [e], number(N), {atom_concat('e',N,TypeVar)}.
typeVariable(tpV(TypeVar)) -->
   [f], number(N), {atom_concat('e',N,TypeVar)}.
singleNumber('0') --> [0].
singleNumber('1') --> [1].
singleNumber('2') --> [2].
singleNumber('3') --> [3].
singleNumber('4') --> [4].
singleNumber('5') --> [5].
singleNumber('6') --> [6].
singleNumber('7') --> [7].
singleNumber('8') --> [8].
singleNumber('9') --> [9].
singleNumber('0') --> [0].
number(SN) --> singleNumber(SN).
number(NUM) --> singleNumber(SN),
   number(N), {atom_concat(SN,N,NUM)}.
```

Untuk tipe komposit, spesifikasi DCG yang dibuat juga mengikuti penjelasan tentang tipe komposit yang dijelaskan pada sub-bab 4.1.1. Berikut ini adalah spesifikasi DCG untuk tipe komposit tersebut:

- Sebuah tipe komposit yang berada di dalam tanda kurung juga merupakan suatu tipe komposit.

```
compositeType(CompType) -->
   ['('], compositeType(CompType), [')'].
```

- Selanjutnya, tipe komposit didefinisikan sebagai urutan dari *type-variable*, tipe komposit lain, ataupun kombinasinya.

```
compositeType(CompType) -->
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
      typeVariable(Tv), ['>'], compEl(Tv,CompType).
   compositeType(CompType) -->
      ['('], compositeType(CompT), [')'],
      ['>'], compEl(CompT,CompType).
   compEl(TpRes,CompType) -->
      typeVariable(Tv), {CompType = tp(TpRes,Tv)}.
   compEl(TpRes,CompType) -->
      ['('], compositeType(CompT), [')'],
      {CompType = tp(TpRes,CompT)}.
   compEl(TpRes,CompType) -->
      typeVariable(Tv), ['>'],
      compEl(Tv,CmpTp), {CompType = tp(TpRes,CmpTp)}.
   compEl(TpRes,CompType) -->
      ['('], compositeType(CompT), [')'],['>'],
      compEl(CompT,CmpTp), {CompType = tp(TpRes,CmpTp)}.
```

9.5 Predikat Pendukung untuk Implementasi Algoritma Pencarian *Type Inhabitant*

Sub-bab ini menjelaskan predikat-predikat pendukung yang digunakan dalam implementasi algoritma pencarian *type inhabitant*. Beberapa predikat ada yang telah *built-in* di dalam PROLOG, namun ada beberapa predikat yang diimplementasikan oleh penulis. Adapun predikat-predikat yang diimplementasikan oleh penulis adalah sebagai berikut:

- `findMetaVar([`*`NfScheme`*`],[`*`ListMetaVar`*`])`. *Unify* [*ListMetaVar*] dengan kumpulan *meta-variable* dari [*NfScheme*].

```
findMetaVar( termVar(_, _), []) :- !.
findMetaVar( metaVar(X, Type), [metaVar(X,Type)] ) :- !.
findMetaVar( abstraction(_, Body), PTM) :- !,
   findMetaVar(Body, PTM).
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
findMetaVar( application(X, Y), PTM) :- !,
   findMetaVar(X, PTM_X), findMetaVar(Y, PTM_Y),
   append(PTM_X, PTM_Y, PTM).
```

- `listMetaVar([ListNFScheme],[ListMetaVar])`. *Unify* [*ListMetaVar*] dengan kumpulan *meta-variable* dari setiap *nf-scheme* anggota [*ListNFScheme*].

```
listMetaVar([], []) :- !.
listMetaVar([AscrMem_H | AscrMem_T],
            [ptm(AscrMem_H, ListMetaVar_H)|ListMetaVar_T]):-
   findMetaVar(AscrMem_H, ListMetaVar_H),
   listMetaVar( AscrMem_T, ListMetaVar_T).
```

- `hasMetaVar([List_ptm])`. Sukses jika [*List_ptm*] setidaknya mengandung satu *nf-scheme* yang masih memiliki *meta-variable*.

```
hasMetaVar( [] ) :- !, \+true.
hasMetaVar( [ptm(_, ListOfMetaVar) | _] ) :-
   ListOfMetaVar \== [], !.
hasMetaVar( [ptm(_, ListOfMetaVar) | PTM_T] ) :-
   ListOfMetaVar == [], !,
   hasMetaVar(PTM_T).
```

- `hasATerm([List_ptm], [Result])`. *Unify* [*Result*] dengan 'true' jika [*List_ptm*] mengandung setidaknya sebuah *term*, atau *unify* [*Result*] dengan 'false' jika [*List_ptm*] tidak mengandung sebuah *term* pun.

```
hasATerm([], 'false') :- !.
hasATerm( [ptm(_, ListOfMetaVar) | _], 'true') :-
   ListOfMetaVar == [], !.
hasATerm( [ptm(_, ListOfMetaVar) | PTM_T], Result) :-
   ListOfMetaVar \== [], !,
   hasATerm(PTM_T, Result).
```

- `isValidAnswer([Answer])`. Sukses jika [*Answer*] merupakan *list* [y] atau [n].

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
isValidAnswer(Answer):-
   Answer \== [y], Answer \== [n], !, \+true.
isValidAnswer(Answer):-
   Answer == [y], !.
isValidAnswer(Answer):-
   Answer == [n], !.
```

- `tail ([Tipe1],[Tipe2])`. *Unify* [Tipe2] dengan *tail* dari [Tipe1].

```
tail(tpV(X),tpV(X)) :- !.
tail(tp(_,B),Tail) :- tail(B,Tail).
```

- `reversedTypeList([Tipe], [`*`ListTipe1`*`], [`*`ListTipe2`*`])`. *Unify* [*ListTipe2*] dengan daftar premis dari [Tipe], dengan susunan terbalik (dalam pemanggilan, [*ListTipe1*] diisi dengan *List* kosong).

```
reversedTypeList(tpV(_), RevList, RevList) :- !.
reversedTypeList(tp(A,B), RevList, RevListResult) :-
   append([A], RevList, NewRevList),
   reversedTypeList(B, NewRevList, RevListResult).
```

- `getSameTypeBasedOnTailAndCollOfType ([`*`MetaVariableNumber`*`], [Tipe], [`*`TipeTail`*`], [`*`ListTpPos1`*`], [`*`ListTpPos2`*`], [Pos])`. *Unify* [*ListTpPos1*] dengan premis dari [Tipe] yang *tail*-nya memiliki tipe yang sama dengan [*TipeTail*]. Kemudian predikat ini juga meng-*unify* [*ListTpPos2*] dengan semua premis dari [Tipe].

```
%Get all type that has a same tail And
%collection of the premises of the type
getSameTypeBasedOnTailAndCollOfType(_, tpV(_),
                                    _, [], [],
                                    _) :- !.
getSameTypeBasedOnTailAndCollOfType(MetaNum, tp(A,B), Tail,
                                    ListOfTypeSameTail,
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
                                            [tpPos(A,PosPlusMetaNum)|
                                            ListOfPremisesOfType],
                                            Pos) :-
    tail(A,TempTail), TempTail \== Tail, !,
    atom_concat(MetaNum, Pos, PosPlusMetaNum),
    NewPos is Pos +1,
    getSameTypeBasedOnTailAndCollOfType(MetaNum, B, Tail,
                                              ListOfTypeSameTail,
                                              ListOfPremisesOfType,
                                              NewPos).
getSameTypeBasedOnTailAndCollOfType(MetaNum, tp(A,B), Tail,
                                            [tpPos(A,PosPlusMetaNum)|
                                            ListOfTypeSameTail],
                                            [tpPos(A,PosPlusMetaNum)|
                                            ListOfPremisesOfType],
                                            Pos) :-
    tail(A,TempTail), TempTail == Tail,  !,
    atom_concat(MetaNum, Pos, PosPlusMetaNum),
    NewPos is Pos +1,
    getSameTypeBasedOnTailAndCollOfType(MetaNum, B, Tail,
                                              ListOfTypeSameTail,
                                              ListOfPremisesOfType,
                                              NewPos).
```

- `getCovAbstSameTp ([`*`nf-scheme`*`],[`*`ListCovAbs`*`],[MetaVar],[Tipe],[Result])`. *Unify* [*ListCovAbs*] dengan tipe *covering abstractor* dari [MetaVar] pada [*nf-scheme*], yang memiliki tipe *tail* yang sama dengan [Tipe].

```
%getCovering Abstractor which has a type that its tail
%has the same type with the given type
getCovAbstSameTp(termVar(_,_), [], _, _, Result) :- !,
    Result = 'Not_Found'.
getCovAbstSameTp(metaVar(X,Type),[],MetaVar,_,Result) :-
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
   MetaVar == metaVar(X,Type), !,
   Result = 'Found'.
getCovAbstSameTp(metaVar(X,Type),[],MetaVar,_,Result) :-
   MetaVar \== metaVar(X,Type), !,
   Result = 'Not_Found'.
getCovAbstSameTp(abstraction(Head,Body), ListCovAbs,
                 MetaVar, Type, AbsResult) :- !,
   getCovAbstSameTp(Body, BodyListCovAbs,
                 MetaVar, Type, BodyResult),
   absDecision2(Head, Type, CovAbs, BodyResult, AbsResult),
   append(CovAbs, BodyListCovAbs, ListCovAbs).
getCovAbstSameTp(application(P,Q), ListCovAbs,
                 MetaVar, Type, Result) :- !,
   getCovAbstSameTp(P, P_ListCovAbs,
                 MetaVar, Type, P_Result),
   getCovAbstSameTp(Q, Q_ListCovAbs,
                 MetaVar, Type, Q_Result),
   appDecision2(P_Result, Q_Result, P_ListCovAbs,
               Q_ListCovAbs, CovAbs, DecsRes),
   Result = DecsRes, ListCovAbs = CovAbs.
absDecision2(termVar(X,Tp), Type, [tpPos(Tp,Num)],
             BodyResult, AbsResult) :-
   BodyResult == 'Found',
   tail(Tp, TailTp),
   Type == TailTp,!,
   atom_concat(x,Num,X),
   AbsResult = 'Found'.
absDecision2(termVar(_,Tp), Type, [], BodyResult,
             AbsResult) :-
   BodyResult == 'Found',
   tail(Tp, TailTp),
   Type \== TailTp,!,
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
    AbsResult = 'Found'.
absDecision2(_, _, [], BodyResult, AbsResult) :-
    BodyResult == 'Not_Found',!,
    AbsResult ='Not_Found' .
appDecision2(P_Result, _, P_ListCovAbs, _,
             P_ListCovAbs, 'Found') :-
    P_Result == 'Found' , !.
appDecision2(_, Q_Result, _, Q_ListCovAbs,
             Q_ListCovAbs, 'Found') :-
    Q_Result == 'Found', !.
appDecision2(P_Result, Q_Result, _, _, [],
             'Not_Found') :-
    Q_Result == 'Not_Found',
    P_Result == 'Not_Found', !.
```

Sedangkan predikat-predikat yang sudah *built-in* dalam PROLOG dan digunakan dalam program ini adalah sebagai berikut:

- `append` /3.
- `name` /2.
- `gensym` /2.
- `reset_gensym` /2.
- `atom_concat` /3.
- `write` /1.

Penjelasan predikat-predikat yang sudah *built-in* dalam PROLOG dapat dilihat pada [Wie07].

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

9.6 Implementasi Algoritma Pencarian *Type Inhabitant*

Sub-bab ini menjelaskan implementasi algoritma pencarian *type inhabitant* yang dijelaskan pada sub-bab 8.7. Dalam implementasi ini, *output* dikeluarkan dengan predikat `write /1` sebagai bagian dari pemrosesan. Implementasi algoritma tersebut dilakukan pada predikat `inhabitant /2`. Struktur dari predikat `inhabitant /2` tersebut adalah sebagai berikut:

```
inhabitant(Tipe, AscriptList).
```

Dalam predikat `inhabitant /2` tersebut, terdapat 2 argumen, yaitu:

1. `Tipe`, merupakan tipe yang diberikan sebagai *input* dari algoritma ini, yang ingin dicari *inhabitant*-nya.
2. `AscriptList`, merupakan himpunan $\mathscr{A}(\tau, d)$ ($d = 0, 1, 2, \ldots$) yang dihasilkan algoritma ini.

Dalam implementasi algoritma pencarian *type inhabitant* ini, predikat `inhabitant /2` memiliki 2 definisi. Satu definisi untuk menangani kondisi ketika tipe yang diberikan atomik. Satu definisi untuk menangani kondisi ketika tipe yang diberikan komposit. Berikut adalah code implementasi predikat `inhabitant /2` (predikat `searchIhbt /3` yang dipanggil dalam predikat ini dijelaskan kemudian):

1. Kasus ketika tipe yang diberikan atomic. Tidak ada *inhabitant* yang dihasilkan.

```
inhabitant(tpV(_), []) :- !,
   write('finish_atom;').
```

2. Kasus ketika tipe yang diberikan komposit. Algoritma ini akan mencari *inhabitant*-nya (jika ada).

```
inhabitant(tp(A,B), AscriptList) :- !,
   gensym(meta,Meta),
   searchIhbt( ascript( tp(A,B), 0,
                [metaVar(Meta, tp(A,B))] ), [],
                AscriptList ).
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

Pemanggilan terhadap Predikat `searchIhbt /3` pada predikat `inhabitant /2` merupakan implementasi langkah 0 algoritma pencarian *inhabitant*. Predikat `searchIhbt /3` akan mencari *inhabitant* untuk tipe yang diberikan. Predikat tersebut merupakan implementasi langkah $\boldsymbol{d+1}$ dari algoritma ini (seperti yang dijelaskan pada sub-bab 8.7). Adapun struktur `searchIhbt /3` adalah sebagai berikut:

```
searchIhbt(Ascript, PrevAscripList, AscriptList).
```

Predikat `searchIhbt /3` memiliki 3 argumen yang dijelaskan sebagai berikut:

- `Ascript`. Himpunan $\mathscr{A}(\boldsymbol{\tau, d})$ (untuk suatu tipe $\boldsymbol{\tau}$ dan bilangan bulat $\boldsymbol{d}$).
- `PrevAscripList`. Hasil kumpulan himpunan $\mathscr{A}(\boldsymbol{\tau, d})$(dengan $\boldsymbol{d = 0, 1, 2, \ldots d-1}$) dari proses pencarian.
- `AscriptList`. Hasil kumpulan seluruh himpunan $\mathscr{A}(\boldsymbol{\tau, d})$($\boldsymbol{d = 0, 1, 2, \ldots}$) dari proses pencarian.

Predikat `searchIhbt /3` memiliki dua definisi yang implementasinya adalah sebagai berikut (predikat `searchIhbt2 /4` yang dipanggil dalam predikat ini dijelaskan kemudian):

1. Menangani kasus ketika $\mathscr{A}(\boldsymbol{\tau, d}) = \boldsymbol{\{\}}$. Ketika menemui kasus ini, algoritma berhenti. Dalam definisi ini, predikat `searchIhbt /3` juga mengeluarkan *output* $\mathscr{A}(\boldsymbol{\tau, d})$ (untuk suatu $\boldsymbol{d}$ ($\boldsymbol{d = 0, 1, 2, \ldots}$)).

```
searchIhbt( ascript( Type, Depth, AscriptMember) ,
            PrevAscripList, AscriptList) :-
   AscriptMember == [], !,
   append(PrevAscripList,[ascript(Type,Depth,AscriptMember)],
          AscriptList),
     write('finish;'),
     generateStringForSeqAscript(AscriptList, Str),
     write(Str).
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

2. Predikat ini mengumpulkan semua *meta-variable* untuk setiap anggota $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$. Kemudian memanggil predikat `searchIhbt2 /4` untuk menentukan apakah $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$ masih memiliki setidaknya satu anggota yang mempunyai *meta-variable* atau tidak. Berdasarkan keputusan tersebut, algoritma akan berhenti atau memulai pembuatan $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d}+\boldsymbol{1})$.

```
searchIhbt( ascript( Type, Depth, AscriptMember),
            PrevAscripList, AscriptList) :- !,
   listMetaVar(AscriptMember, ListPTM),
   searchIhbt2(ascript( Type, Depth, AscriptMember) , ListPTM,
               PrevAscripList, AscriptList).
```

Predikat `searchIhbt2 /4` memiliki dua definisi. Satu definisi untuk menangani kasus ketika tidak ada lagi anggota $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$ yang memiliki *meta-variable* (pada kasus ini, algoritma akan berhenti). Satu definisi untuk memulai pembuatan $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d}+\boldsymbol{1})$, yaitu ketika masih ada anggota $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$ yang memiliki *meta-variable*. Berikut adalah implementasi predikat `searchIhbt2 /4`(predikat `searchIhbt3 /5` yang dipanggil dalam predikat ini dijelaskan kemudian):

1. Kasus ketika tidak ada lagi anggota $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$ yang memiliki *meta-variable*. Pada kasus ini, algoritma berhenti. Dalam definisi ini, predikat `searchIhbt2 /4` juga mengeluarkan *output* $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$ (untuk suatu $\boldsymbol{d}$ ($\boldsymbol{d} = \boldsymbol{0}, \boldsymbol{1}, \boldsymbol{2}, \ldots$)).

```
searchIhbt2(ascript( Type, Depth, AscriptMember),
                              ListPTM, PrevAscripList, AscriptList) :-
   \+ hasMetaVar(ListPTM),
   !,
   append( PrevAscripList, [ascript( Type, Depth, AscriptMember )],
           AscriptList),
   write('finish;'),
   generateStringForSeqAscript(AscriptList, Str),
   write(Str).
```

2. Kasus ketika masih ada anggota $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$ yang memiliki *meta-variable*. Selanjutnya dicek apakah $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$ sudah memiliki minimal satu *inhabitant* atau tidak

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

dengan predikat `hasATerm /2`. Setelah itu dipanggil predikat `searchIhbt3 /5` untuk melanjutkan pemrosesan.

```
searchIhbt2(Ascript, ListPTM, PrevAscripList, AscriptList) :-
    hasMetaVar(ListPTM), !,
    hasATerm(ListPTM, HasATerm),
    searchIhbt3(HasATerm, Ascript, ListPTM,
                PrevAscripList, AscriptList).
```

Predikat `searchIhbt3 /5` memiliki dua definisi. Berikut adalah implementasi predikat tersebut:

1. Kasus ketika $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$ sudah memiliki setidaknya satu *inhabitant*. Dalam kasus ini, program akan bertanya pada pengguna apakah mau melanjutkan pencarian *inhabitant* pada nilai d berikutnya, atau berhenti. Selanjutnya jawaban dari pengguna diproses oleh predikat `processRetrialAnswer /5` (dijelaskan kemudian). Berikut adalah implementasi predikat `searchIhbt3 /5` tersebut.

```
searchIhbt3(HasATerm,Ascript,ListPTM,PrevAscripList,AscriptList):-
    HasATerm == 'true', !,
    append(PrevAscripList, [Ascript], NewAscriptList),
    write('finish_partially;'),
    generateStringForSeqAscript(NewAscriptList, Str),
    write( Str), write(';'),
    write('arg1('), write(Ascript), write(').\n'),
    write('arg2('), write(ListPTM), write(').\n'),
    write('arg3('), write(PrevAscripList), write(').\n'),
    write('arg4('), write('AscriptMember'), write(').\n').
```

2. Berikut ini adalah kasus kedua dari implementasi predikat `searchIhbt3 /5`, yaitu kasus ketika $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$ belum memiliki satu *inhabitant* pun. Pada kasus ini program mencoba untuk membuat *suitable replacement* untuk setiap *meta-variable* dan membuat $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d}+\mathbf{1})$. Untuk melakukan hal tersebut ada dua predikat yang berperan, `constructSuitableReplacement /3` dan `applyReplacement /2` (dijelaskan kemudian). Berikut adalah implementasi predikat `searchIhbt3 /5`.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
searchIhbt3(HasATerm, ascript( Type, Depth, AscriptMember) ,
            ListPTM, PrevAscripList, AscriptList) :-
   HasATerm == 'false', !,
   append(PrevAscripList, [ascript(Type, Depth, AscriptMember)],
          NewAscriptList),
   constructSuitableReplacement(Type, ListPTM, ListPTMSR),
   applyReplacement(ListPTMSR, NewAscriptMember),
   NewDepth is Depth + 1,
   searchIhbt(ascript(Type, NewDepth, NewAscriptMember) ,
              NewAscriptList, AscriptList).
```

Berikut adalah penjelasan predikat `constructSuitableReplacement /3`. Predikat ini membuat *suitable replacement* untuk setiap *meta-variable* pada setiap *nf-scheme* anggota $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$. `constructSuitableReplacement /3` memiliki struktur sebagai berikut:

`constructSuitableReplacement([Tipe], [`*`List_ptm`*`], [`*`List_ptmsr`*`])`,

dan tiga argumen yaitu:

- `[Tipe]`. Merupakan tipe $\boldsymbol{\tau}$ dari $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$.
- `[`*`List_ptm`*`]`. Merupakan daftar pasangan *nf-scheme* dan setiap *meta-variable* didalamnya. Daftar ini direpresentasikan dengan *list* dari `ptm` (struktur `ptm` dijelaskan pada 9.1.6).
- `[`*`List_ptmsr`*`]`. Merupakan daftar pasangan *nf-scheme* dan *suitable replacement* untuk setiap *meta-variable* didalamnya. Daftar ini direpresentasikan dengan *list* dari `ptmsr` (struktur `ptmsr` dijelaskan pada 9.1.5).

Dalam membuat *suitable replacement* untuk *list* dari `ptm`, Predikat ini membuat *suitable replacement* satu persatu untuk setiap `ptm` dengan memanggil predikat `constructSuitableReplacementForAPTM /2` (dijelaskan kemudian). Berikut adalah implementasi predikat `constructSuitableReplacement /3`:

```
%-----------------------------------------------------------------------
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
% Part of SubStep II
% try to apply IIa-IIb to each of proper Nf-schemes in A(t,d)
% Construct suitable replacement from a list of ptm.
%------------------------------------------------------------------------
   constructSuitableReplacement( _, [], [] ) :- !.
   constructSuitableReplacement( Type,
                                 [ListPTM_Head | ListPTM_Tail] ,
                                 ListPTMSR) :-
      constructSuitableReplacementForAPTM( ListPTM_Head, PTMSR_H),
      constructSuitableReplacement( Type, ListPTM_Tail, PTMSR_T),
      appendPTMSR(PTMSR_H, PTMSR_T, ListPTMSR).
    appendPTMSR(PTMSR_H, PTMSR_T, ListPTMSR) :-
      PTMSR_H == [], !,
      append(PTMSR_H, PTMSR_T, ListPTMSR).
   appendPTMSR(PTMSR_H, PTMSR_T, ListPTMSR) :-
      PTMSR_H \== [], !,
      append([PTMSR_H], PTMSR_T, ListPTMSR).
```

Berikut adalah implementasi predikat `constructSuitableReplacementForAPTM /2`.

```
%------------------------------------------------------------------------
%SubsubStep IIa
%Construct suitable replacement from a ptm
%------------------------------------------------------------------------
   constructSuitableReplacementForAPTM(ptm(Term, ListMetaVar),PTMSR):-
      constructSuitRepForListOfMetaVar(Term, ListMetaVar, ListSR),
      %subsubstep IIb - decide rejection
      decideReject(Term, ListMetaVar, ListSR, PTMSR).
   constructSuitRepForListOfMetaVar(_, [], []) :- !.
   constructSuitRepForListOfMetaVar(Term,
                                    [ListMetaVar_H | ListMetaVar_T ],
                                    ListSR) :-
      constructSuitRepForAMetaVar(Term, ListMetaVar_H, SuitRep),
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
      constructSuitRepForListOfMetaVar(Term, ListMetaVar_T, SuitRep_T),
      append([SuitRep], SuitRep_T, ListSR).
```

Predikat tersebut membuat *suitable replacement* untuk sebuah `ptm`. Predikat ini juga mengimplementasikan subsublangkah IIb (bagian awal) dari algoritma pencarian *type inhabitant*, yaitu melakukan *reject* untuk *nf-scheme* yang memiliki satu atau beberapa *meta-variable*, yang tidak memiliki *suitable replacement*. Hal tersebut ditangani predikat `decideReject /4`. Berikut adalah implementasi predikat `decideReject /4`:

```
%------------------------------------------------------------------------
%SubsubStep IIb
%------------------------------------------------------------------------
   decideReject(Term, ListMetaVar, ListSR, PTMSR) :-
      ListMetaVar == [], !,
      PTMSR = ptmsr(Term, ListSR).
   %Reject if there are no more suitable replacement for
   decideReject(Term, ListMetaVar, ListSR, PTMSR) :-
      ListMetaVar \== [],
      %check whether all Meta Variable has a suitable replacement not
      hasSR(Term, ListSR, ListSR, PTMSR).
   %All meta variable has suitable replacement
   hasSR(Term, ListSR, [], PTMSR) :- !,
      PTMSR = ptmsr(Term, ListSR).
    %there's a meta variable that has no suitable replacement
   hasSR(_, _, [sr(_, SuitableReplacement)| _], PTMSR) :-
      SuitableReplacement == [], !,
      PTMSR = [].
   hasSR(Term, ListSR, [sr(_,SuitableReplacement)|ListSR_T],PTMSR):-
      SuitableReplacement \== [], !,
      hasSR(Term, ListSR, ListSR_T, PTMSR).
%------------------------------------------------------------------------
%End of SubsubStep IIb
%------------------------------------------------------------------------
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

Selanjutnya, berikut ini adalah predikat `constructSuitRepForAMetaVar /3` yang berfungsi untuk membuat *suitable replacement* untuk sebuah *meta-variable*. Dalam bagian ini diimplementasikan bagian IIa1 dan IIa2 dari algoritma pencarian *type-inhabitant*.

```
%----------------------------------------------------------------------------
% Part IIa1 & IIa2
% construct suitable replacement for a Meta Variable
% it will be divided into two part, IIa1 & IIa2
%----------------------------------------------------------------------------
   constructSuitRepForAMetaVar( Term, metaVar(X, Type), SuitRep):-
      tail(Type, TailFromType),
      atom_concat(meta,MetaNum,X),
      getSameTypeBasedOnTailAndCollOfType(MetaNum, Type, TailFromType,
                                          ListTypeHasSameTail,
                                          ListOfPremisesOfType, 1),
      %this step below will decide whether we should apply IIa1 or IIa2,
      %then apply the chosen rule.
      constSuitRep(Term, metaVar(X, Type), SuitRep,
                ListTypeHasSameTail, ListOfPremisesOfType).
%----------------------------------------------------------------------------
%Part IIa1
%----------------------------------------------------------------------------
   constSuitRep(Term, metaVar(X, Type), SuitRep,
             ListTypeHasSameTail, ListOfPremisesOfType) :-
      ListTypeHasSameTail \== [], !,
      %Define IIa1.
         defineSuitRep(ListTypeHasSameTail,
                    ListOfPremisesOfType, ListSuitRep1),
      %apply IIa2
         tail(Type, TailTypeMetaVar),
         getCovAbstSameTp(Term, ListCovAbsType,
```

```
                    metaVar(X, Type), TailTypeMetaVar, _),
      %Define IIa2
        defineSuitRep(ListCovAbsType,
                      ListOfPremisesOfType, ListSuitRep2),
        append(ListSuitRep1, ListSuitRep2, ListSuitRep),
    SuitRep = sr(metaVar(X, Type), ListSuitRep).
%------------------------------------------------------------------------
% Part IIa2
% applied where m = 0
%------------------------------------------------------------------------
   constSuitRep(Term, metaVar(X, tpV(Z)),
                SuitRep, _, ListOfPremisesOfType) :- !,
      tail(tpV(Z), TailTypeMetaVar),
      getCovAbstSameTp(Term, ListCovAbsType,
                       metaVar(X, tpV(Z)), TailTypeMetaVar, _),
    %Define IIa2.
      defineSuitRep(ListCovAbsType,ListOfPremisesOfType,ListSuitRep),
      SuitRep = sr(metaVar(X, tpV(Z)), ListSuitRep).

%------------------------------------------------------------------------
% Part IIa2
% applied where there's no premises that has the same tail
%------------------------------------------------------------------------
   constSuitRep(Term, metaVar(X, Type),
                SuitRep, ListTypeHasSameTail, ListOfPremisesOfType) :-
      ListTypeHasSameTail == [], !,
      tail(Type, TailTypeMetaVar),
      getCovAbstSameTp(Term, ListCovAbsType,
                       metaVar(X, Type), TailTypeMetaVar, _),
    %Define IIa2.
      defineSuitRep(ListCovAbsType,ListOfPremisesOfType,ListSuitRep),
      SuitRep = sr(metaVar(X, Type), ListSuitRep).
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

Berikut ini adalah implementasi predikat `applyReplacement /2`. Predikat ini berfungsi untuk menerapkan *suitable replacement* yang telah dihasilkan oleh predikat `constructSuitableReplacement /3`. Dengan kata lain, predikat ini adalah implementasi subsublangkah IIb (bagian akhir) dari algoritma pencarian *type inhabitant.*

```
applyReplacement([],[]) :- !.
applyReplacement([ptmsr(NFScheme,ListOfSR)|ListPTMSR_T],
                 ListNewNFScheme):- !,
   applyReplacementToAPTMSR([NFScheme],ListOfSR,ListNewNFScheme_H),
   applyReplacement(ListPTMSR_T, ListNewNFScheme_T),
   append(ListNewNFScheme_H, ListNewNFScheme_T, ListNewNFScheme).
applyReplacementToAPTMSR(ListNewNFScheme, [],
                         ListNewNFScheme) :- !.
applyReplacementToAPTMSR(CurrListNFScheme,
                         [sr( MetaVar, ReplacementTermList) |
                          ListOfSR_T],
                         ListNewNFScheme) :- !,
   applyAnSRtoNFScheme(CurrListNFScheme, MetaVar,
                    ReplacementTermList, ListNFSchemeFromAnSR),
   applyReplacementToAPTMSR(ListNFSchemeFromAnSR,
                            ListOfSR_T, ListNewNFScheme).
%Replace a Meta Variable in List of NF Scheme with ALL
%suitable replacement it will construct a LIST of new NF Scheme
applyAnSRtoNFScheme(_, _, [], []) :- !.
applyAnSRtoNFScheme(ListNFScheme, MetaVar,
                    [ReplacementTermList_H | ReplacementTermList_T],
                    NewListOfNFScheme) :- !,
   replaceMetaVarInNFScheme(ListNFScheme, MetaVar,
                            ReplacementTermList_H, NewNFScheme),
   applyAnSRtoNFScheme(ListNFScheme, MetaVar,
                    ReplacementTermList_T, ListNewNFScheme),
   append(NewNFScheme, ListNewNFScheme, NewListOfNFScheme).
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
replaceMetaVarInNFScheme([], _, _, []) :- !.
replaceMetaVarInNFScheme([NFScheme | ListNFScheme_T],
                         MetaVar, ReplacementTerm,
                         [NewNFScheme|NewNFScheme_T]) :-
   replaceMetaVarInAnNFScheme(NFScheme, MetaVar,
                              ReplacementTerm, NewNFScheme),
   replaceMetaVarInNFScheme(ListNFScheme_T, MetaVar,
                            ReplacementTerm, NewNFScheme_T).
%Replace a Meta Variable in NF Scheme with A suitable replacement
%it will construct a new NF Scheme
replaceMetaVarInAnNFScheme(termVar(X,Type), _, _,
                           termVar(X,Type)) :- !.
replaceMetaVarInAnNFScheme(metaVar(X,Type), metaVar(Y, Type2),
                           ReplacementTerm, NewNFScheme) :-
   X == Y, Type == Type2, !,
   NewNFScheme = ReplacementTerm.
replaceMetaVarInAnNFScheme(metaVar(X,Type), metaVar(Y, _), _,
                           NewNFScheme) :-
   X \== Y, !,
   NewNFScheme = metaVar(X,Type).
replaceMetaVarInAnNFScheme(metaVar(X,Type), metaVar(_, Type2), _,
                           NewNFScheme) :-
   Type \== Type2, !,
   NewNFScheme = metaVar(X,Type).
replaceMetaVarInAnNFScheme(abstraction(Head,Body), MetaVar,
                           ReplacementTerm,
                           abstraction(Head,NewNFScheme)) :- !,
   replaceMetaVarInAnNFScheme(Body, MetaVar, ReplacementTerm,
                              NewNFScheme).
replaceMetaVarInAnNFScheme(application(X,Y), MetaVar,
                           ReplacementTerm,
                           application(NewNFScheme_X ,
```


Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008

```
                                NewNFScheme_Y)) :- !,
    replaceMetaVarInAnNFScheme(X, MetaVar, ReplacementTerm,
                                  NewNFScheme_X),
    replaceMetaVarInAnNFScheme(Y, MetaVar, ReplacementTerm,
                                  NewNFScheme_Y).
```

Selain semua predikat yang telah dijelaskan, terdapat satu predikat lagi yang penting. Predikat tersebut digunakan untuk melakukan *query* kembali, untuk mencari *inhabitant* berikutnya berdasarkan informasi pencarian *inhabitant* yang telah dihasilkan sebelumnya. Dalam program ini, predikat tersebut digunakan untuk mencari *inhabitant* berikutnya pada kasus ketika satu *inhabitant* telah ditemukan namun pengguna masih ingin mencari *inhabitant* lagi. Predikat tersebut adalah `goFindNextInhabitant /1`. Berikut adalah struktur predikat tersebut:

```
goFindNextInhabitant([FileOfArg]).
```

[FileOfArg] merupakan file yang berisikan argumen untuk pencarian *inhabitant berikutnya.* File ini berisikan informasi argumen yang dibutuhkan oleh predikat `findNextInhabitant /4` untuk mencari *inhabitant* berikutnya. argumen tersebut adalah:

- `Ascript`. Himpunan $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$ (untuk suatu tipe $\boldsymbol{\tau}$ dan bilangan bulat $\boldsymbol{d}$). Di dalam file tersebut argumen ini dibungkus oleh *functor* `arg1 /1`.
- `ListPTM`. *List* dari `ptm`. Di dalam file tersebut argumen ini dibungkus oleh *functor* `arg2 /1`.
- `PrevAscripList`. Hasil kumpulan himpunan $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$(dengan $\boldsymbol{d = 0, 1, 2, \ldots d-1}$) dari proses pencarian. Di dalam file tersebut argumen ini dibungkus oleh *functor* `arg3 /1`.
- `AscriptList`. Hasil kumpulan seluruh himpunan $\mathscr{A}(\boldsymbol{\tau}, \boldsymbol{d})$($\boldsymbol{d = 0, 1, 2, \ldots}$) dari proses pencarian. Di dalam file tersebut argumen ini dibungkus oleh *functor* `arg4 /1`.

Berikut ini adalah implementasi predikat `goFindNextInhabitant /1` tersebut:

```
goFindNextInhabitant(FileOfArg) :-
    see(FileOfArg),
    read(A),read(B),read(C),read(D),
    findNextInhabitant(A,B,C,D),
    seen.
```

Dalam implementasi predikat tersebut, terdapat pemanggilan kepada predikat `findNextInhabitant /4` yang berfungsi untuk memulai pencarian *inhabitant*. Berikut adalah implementasinya:

```
findNextInhabitant(arg1(ascript( Type, Depth, AscriptMember)),
                   arg2(ListPTM),
                   arg3(PrevAscripList),
                   arg4(AscriptList)):-
   append(PrevAscripList, [ascript( Type, Depth, AscriptMember )],
          NewAscriptList),
   constructSuitableReplacement( Type, ListPTM, ListPTMSR),
   applyReplacement(ListPTMSR, NewAscriptMember),
   NewDepth is Depth + 1,
   searchIhbt(ascript(Type, NewDepth, NewAscriptMember),
             NewAscriptList, AscriptList).
```

9.7 Uji Coba Hasil Implementasi Algoritma Pencarian Type Inhabitant

Setelah implementasi, dilakukan ujicoba pada 33 tipe. Hasilnya, untuk setiap tipe yang diberikan dalam ujicoba tersebut, program implementasi algoritma pencarian *type inhabitant* ini mengeluarkan *inhabitant* yang tepat. Hasil uji coba ini dirangkum dan disajikan pada Lampiran E.

Implementasi algoritma principal...,Ario Santoso, FASILKOM UI, 2008