自然言語処理 93−11
(1993. 1. 14)

タイプ付きユニフィケーション文法の新しい基礎

赤間 清

北海道大学 工学部 情報工学科

タイプ付きユニフィケーション文法 (TUG) は、ユニフィケーション文法の拡張であり、強力な表現力を持ち、計算機言語学や自然言語処理にとって有用である。本論文では、TUG を**一般化論理プログラム（GLP）の理論**を基礎にして定式化する。GLP の理論は、Specialization System と呼ばれるプログラムの基礎空間 X と、その上の論理プログラム P をパラメータとして持つ**スキーマ理論** GLP(X,P) であり、X と P を変えることによって、種々の宣言的知識表現（個々の文法など）を統一的に説明することができる。本論文では、TUG が実は GLP のサブクラスであることを例を用いて示す。これによって、TUG に対して、その**宣言的意味論**が与えられ、**SLD Resolution** や**プログラム変換**などの処理が明快に行なえ、推論の処理系を容易に構築できるようになる。

A New Foundation of Typed Unification Grammars

Kiyoshi Akama

Faculty of Engineering, Hokkaido University

Typed unification grammar (TUG) is an extension of **unification grammar** and provides linguists with a more expressive power integrating disjunction, conjunctions and conditional expressions of typed feature structures. This paper shows that TUGs can be regarded as **generalized logic programs** (GLPs). GLP theory is a schema in the sense that it has two parameters X (a base space called a specialization system) and P (a logic program on the base space). Each instance of various declarative knowledge representations including grammars is regarded as a GLP by constructing a specialization system X and a logic program P for it. In this paper we show that TUG is a subclass of GLP. This enables us to apply to each TUG the general theory of **declarative semantics, SLD resolution, program transformation** for GLPs. It also means that we can easily construct TUG inference systems using GLP-based programming language UL/α.

1 はじめに

タイプ付きユニフィケーション文法（TUG）は、ユニフィケーション文法の拡張であり、**素性構造にタイプ**を付けることによって、言語学上のさまざまなレベルの制約を**モジュラー性**高く表現することを可能にする。TUG は、計算機言語学や自然言語処理を高度化するために重要な枠組と考えられるが、その理論的基礎は まだ十分に解明されたとはいえず [4]、TUG の推論を行なう処理系を作成する技術にも不明の点が少なくない。

本論文では、TUG を**一般化論理プログラム（GLP）の理論** [1, 2, 3] を基礎にしてとらえる。GLP の理論は、Specialization System と呼ばれるプログラムの基礎空間 X とその上の論理プログラム P をパラメータとして持つ**スキーマ理論** GLP(X,P) であり、X と P を変えることによって、種々の宣言的知識表現（個々の文法など）を統一的に説明することができる。

本論文では、TUG が実は GLP のサブクラスであることを例を用いて示す。これによって、TUG に対して、その**宣言的意味論**が与えられ、**SLD Resolution** や**プログラム変換**などの処理が、厳密な理論的基礎をもって、明快に行なえることになる。また、TUG の処理系は、GLP の理論を体現するプログラミング言語 **UL/α** を用いれば、簡単に構築できる。GLP と UL/α の枠組は、TUG の枠組をさらに拡張していくためにも有用である。

本論文では、紙面の都合上、理論を記述することは省略し、TUG で書かれた 1 つの文法 g [6]（この例が言語学的に見て TUG にふさわしいか否かはここでは問わない）がどのようにして GLP とみなせるかを示す。次の記述は、g の文法規則の一部である。

$$
\begin{array}{l}
LEX\_CAT = DET \vee PN \vee V \vee N. \\
SENTENCE = \\
[STRING: \boxed{\text{str}}, C\_STR: S[NP: \boxed{\text{np}}, VP: \boxed{\text{vp}}]] :- \\
NOUN\_PH[STRING: \boxed{\text{npstr}}, C\_STR: \boxed{\text{np}}], \\
VERB\_PH[STRING: \boxed{\text{vpstr}}, C\_STR: \boxed{\text{vp}}], \\
APPEND[F: \boxed{\text{npstr}}, B: \boxed{\text{vpstr}}, W: \boxed{\text{str}}].
\end{array}
$$

g を GLP とみなすために、g に対応する specialization system X とその上の論理プログラム P を与え、g = GLP(X,P) とする。これらのパラメータが決まれば、GLP の理論の一般論 [1, 2, 3] が適用できて、宣言的意味論や**ユニフィケーション**が一意に定まり、さらに推論やプログラム変換などができるようになる。本論文では、g に対するユニフィケーションや推論が、GLP(X,P) に対するユニフィケーションや SLD resolution によって行なえることも述べる。

2 TUG のための Specialization System

2.1 TUG のためのタイプの階層

TUG では、すべてのオブジェクトはタイプを持つ。タイプは階層をなす。本論文の例に用いるタイプの階層を以下に与える。ここでは、

<親タイプ> ::= <子タイプ> ; <子タイプ> ; ···

というシンタックスを用いる。また、同一の親タイプを持つ子タイプ同士の背反性を仮定している。

```
ALL     ::= BASIC; TUG                          verb    ::= i_verb; t_verb
BASIC   ::= LIST; APPEND                        PCAT    ::= S; NP; VP
LIST    ::= NIL; CONS                           NP      ::= NP1; NP2
TUG     ::= WORD; PHRASE; LEX; PCAT; LCAT; SP   LCAT    ::= DET; PN; V; N
WORD    ::= Mary; likes; all; men; ...          V       ::= IV; TV
```

```
PHRASE  ::= sentence; noun_ph; verb_ph          IV      ::= LIVE; GO
noun_ph ::= noun_ph1; noun_ph2                  TV      ::= LOVE; LIKE
LEX     ::= name; verb; det; noun               N       ::= MAN; WOMAN
```

ALL を最も一般的なタイプとして使う。BASIC の中に、プログラミングで基本的なものをいれておく。通常は、LIST はデータ構造、APPEND は述語として扱われるが、ここでは両者ともタイプであり、LIST オブジェクトや APPEND オブジェクトを構成することになる。LIST は、NIL か CONS である。次に TUG に必要なタイプを導入する。WORD は文を構成する単語からなる。PHRASE は、文、名詞句、動詞句からなる。名詞句は 2 種類考える。LEX は辞書項目に対応し、名前、動詞、冠詞、名詞からなる。動詞には自動詞と他動詞がある。PCAT は、文、名詞句、動詞句のデータを扱う。LCAT は、辞書項目に対応するデータを扱う。

2.2 TUG のための論理オブジェクト

TUG のための論理オブジェクトを定義する。まず、変数と素性という 2 種類の記号を導入する。タイプと変数と素性のそれぞれがつくる集合は、互いに背反であると仮定する。

```
<変数>   ::= A; B; C; X; Y; Z; W; ...
<素性>   ::= first; rest; string; f; b; w; c_str; name; arg; ...
```

タイプと変数と素性から、次のようにして、再帰的に対象を定義する。

```
<対象>   ::= [<変数>:<タイプ>] ; [<変数>:<タイプ> {<素性>=<対象>,...}]
```

このとき、たとえば、次のA, B, C は対象である。

```
A = [X:man]
B = [Y:verb_ph {string = [Z:WORD], c_str = [V:ALL]}]
C = [Z:APPEND {f = [P:WORD], b = [Q:WORD], w = [R:WORD]}]
```

上記の定義の中に出てくる < 素性 >=< 対象 > を、素性対象ペアと呼ぶ。また、対象をオブジェクトとも呼ぶ。オブジェクトは、GLP の理論の公理で厳密に規定された論理的な概念である。それで オブジェクトを論理オブジェクトと呼ぶことがある。オブジェクトは必ず変数をともない、それらは 1 対 1 に対応する。オブジェクトが、その先頭に、ある変数X を持つ時、そのオブジェクトはその変数X に対応するという。

変数名が重要でない場合、変数名と ":" を省略できると約束する。たとえば、[X:man] は、[man] と書くことがある。また、[Z:APPEND {f = [P:WORD]}] は、Z と P の両方の名前が重要でないとき、それらを省略すれば、[APPEND {f = [WORD]}] となる。これらはあくまでも記述の省略形であって、理論的には、すべての対象に変数が対応していることに注意したい。スペースの都合上、ある種の LIST に対しても別の略記法を使う。すなわち、たとえば、

```
[CONS {first = [all], rest = [CONS {first = [men], rest = [NIL]}]}]
```

のような LIST オブジェクトを<all men> と書く。

2.3 論理オブジェクトに対する制限

論理オブジェクトには、次の 2 つの条件を課す。

- 任意のオブジェクトに対して、同じ素性を持つ素性対象ペアは唯一である。
- 同じ変数に対するオブジェクトは、まったく等しい。

以下はこの条件を満たさない例である。

```
A = [X:man {age = 12, age = 43}]
B = [Y:verb_ph {string = [Z:WORD], c_str = [Z:ALL]}]
```

2.4 TUG のための基本的な Specialization

基本的な specialization は次の 3 種である。

(S1) [<変数> ⇒ <タイプ>]

(S2) [<変数>/<変数>]

(S3) [<変数>:<素性>=<対象>]

(S1) の specialization は、変数で示された 対象のタイプを subtype に変更する。たとえば、[X:LIST] は、 specialization[X ⇒ CONS] によって、[X:CONS] に変化する。CONS は LIST の subtype だからこの適用が可能である。specialization[X ⇒ CONS] は [X:APPEND] に適用できない。それは、CONS は APPEND とは背反であり、subtype ではないからである。

(S2) の specialization は、対象の変数を変更する。たとえば、specialization[X/Y] は、[X:LIST] を [Y:LIST] に変える。

(S3) の specialization は、変数で示された 対象に 素性対象ペアを追加する。たとえば、[X:LIST] は、specialization[X:first=[Y:WORD]] によって、[X:LIST {first=[Y:WORD]}] に変わる。

2.5 オブジェクトに対する制約

たとえば、NIL でない LIST (つまり CONS) が必ず first と rest を持ち、rest は LIST である。このように、各 タイプは、それに固有の 素性対象ペアを持つ。そのような制約を表現するために、各タイプに 1 つづつ素性対象ペアの集合を宣言する。たとえば、CONS の場合の制約は、

```
CONS ==> {first = [WORD], rest = [LIST]}
```

と表現される。first が [WORD] となっているのは、ここでの CONS がWORD のLIST に対応するものだけを考えているからである。もし、ALL に属する任意のもののLIST を許すなら、

```
CONS ==> {first = [ALL], rest = [LIST]}
```

とすればよい。LIST にはNIL が含まれるので、LIST には必ず持つべき素性はない。したがって、LIST への制約は、LIST ==> {} となる。

本論文の例に用いられる制約のうち、いくつかを以下に示す。

```
PHRASE ==> {string = [LIST]}          verb_ph ==> {c_str = [VP]}
sentence ==> {c_str = [S]}            name ==> {c_str = [PN]}
noun_ph1 ==> {c_str = [NP1]}          det ==> {c_str = [DET]}
noun_ph2 ==> {c_str = [NP2]}          NP2 ==> {name = [PN], agr = [sg]}
```

変数を陽に使わないと書けない制約もある。たとえば、

```
S ==> {np = [NP {agr = [X:SP]}], vp = [VP {agr = [X:SP]}]}
NP1 ==> {det = [DET {agr = [X:SP]}], noun = [N {agr = [X:SP]}], agr = [X:SP]}
```

は、数の一致を表現している。また、

```
VP ==> {v = [TV {agr = [X:SP]}], np = [NP], agr = [X:SP]}
```

は、数の一致のための情報を、下部構造と上部構造で等しくする制約である。最後に、

```
LEX ==> {string = [LIST {first = [X:WORD], rest = [NIL]}],
         c_str = [LCAT {word = [X:WORD]}]}
```

は、表層構造 (string) と c_str を関係づけている。

2.6 タイプ階層による制約の継承

これらの制約は、タイプ階層のもとで解釈される。すなわち、階層の上位のタイプで与えられた制約は下位に継承されるので、下位の タイプでは記述の省略が可能である。逆に、ある タイプに対する真の制約は、その タイプから上位のタイプの制約をすべて集めたものである。たとえば、sentence の上位タイプは、sentence 、PHRASE 、TUG 、ALL であるが、TUG と ALL の制約は{} であり、sentence は {c_str = [S]}、 PHRASE は {string = [LIST]} であるので、それらをあわせて、sentence ==> {string = [LIST], c_str = [S]} となる。そのほかの例を以下に示す。

```
noun_ph ==>  {string = [LIST]}
noun_ph1 ==>  {string = [LIST], c_str = [NP1]}
noun_ph2 ==>  {string = [LIST], c_str = [NP2]}
name ==>  {string = [LIST {first = [X:WORD], rest = [NIL]}],
           c_str = [PN {word = [X:WORD]}]}
```

2.7 実際の Specialization

TUG のための specialization system における実際の specialization は、先に示した基本 specialization に制約に基づく変更を追加したものである。制約に基づく変更とは、基本 specialization を適用した結果が制約を満たさなくなる場合に制約を満たすようにするために (最小限の) 変更を加えるものである。ただし、制約に基づく変更は、素性対象ペアが 1 つ以上存在するオブジェクトに対してだけなされる。[<変数> : <タイプ>] の形のオブジェクトは、素性対象ペアに言及しない概念的表現とみなし、制約を満たしていると考える。制約に基づく変更も基本 specialization と考え、基本 specialization の合成で実際の specialization が与えられるものと考える。

(S1') [<変数> ⇒ <タイプ>] + 制約に基づく変更

(S2') [<変数>/<変数>] + 制約に基づく変更

(S3') [<変数>:<素性>=<対象>] + 制約に基づく変更

(S1') の [<変数> ⇒ <タイプ>] によって、変数で示された 対象のタイプを変更すると、たとえば、[X:LIST] は、specialization[X ⇒ CONS] によって、[X:CONS] に変化するが、これに対しては制約に基づく変更はない。[X:PHRASE {string = [Y:LIST]}] は、基本 specialization[X ⇒ noun_ph1] によって、[X:noun_ph1 {string = [Y:LIST]}] に変わるが、これは、noun__ph1 に対する制約:

```
noun_ph1 ==>  {string = [LIST], c_str = [NP1]}
```

によって、さらに[X:noun_ph1 {string = [Y:LIST], c_str = [NP1]}] へと変化する。

(S2') の [<変数>/<変数>] によって 対象の 変数が変更された場合、タイプの異なる 2 つのオブジェクトが同一変数に対応してしまうことがありうる。これを正しいオブジェクトの形に直す過程で制約違反が起こった場合、やはり制約に基づく変更が必要となる。

(S3') の [<変数>:<素性>=<対象>] によって対象に 素性対象ペアを追加するとき、上と同様の制約違反が起こり、制約に基づく変更が必要となる場合がある。

3 TUG のための論理オブジェクトの単一化

3.1 Unifier の定義

単一化は、GLP の最も基本的な計算である。ここでは簡略化した unifier の定義を与える。詳しくはたとえば、[3] を参照せよ。

定義 1 x と y の unifier とは, $x\theta = y\sigma$ を満たす specialization のペア (θ, σ) のことである。

3.2 TUG のための論理オブジェクトに対する単一化

ここでは、次の (a1) と (b1) を単一化する例を与える。この単一化は、TUG の意味での単一化に対応した結果を与えている。

```
(a1)  [X:noun {string = [CONS {first = [men], rest = [NIL]}]}]
(b1)  [Y:LEX {c_str = [Z:LCAT]}]
```

はじめに、(a1) に specialization を作用させる。2 つの基本 specialization:

```
(p1)   [X/Y]
(p2)   noun ==> {string = [CONS {first = [Z:ALL], rest = [NIL]}],
                 c_str = [N {word = [Z:ALL]}]}
```

をそれぞれ、θ_1,θ_2 とする。また、

```
(a2)  [Y:noun {string = [CONS {first = [men], rest = [NIL]}]}]
(a3)  [Y:noun {string = [CONS {first = [Z:men], rest = [NIL]}],
               c_str = [N {word = [Z:men]}]}]
```

とすれば、(a1)θ_1 = (a2), (a2)θ_2 = (a3) である。
次に、(b1) に specialization を作用させる。3 つの基本 specialization:

```
(q1)  [Y ⇒ noun]
(q2)  noun ==> {string = [CONS {first = [Z:ALL], rest = [NIL]}],
                c_str = [N {word = [Z:ALL]}]}
(q3)  [Z ⇒ men]
```

をそれぞれ、σ_1,σ_2,σ_3 とし、

```
(b2)  [Y:noun {c_str = [Z:LCAT]}]
(b3)  [Y:noun {string = [CONS {first = [Z:ALL], rest = [NIL]}],
               c_str= = [N {word = [Z:ALL]}]}]
(b4)  [Y:noun {string = [CONS {first = [Z:men], rest = [NIL]}],
               c_str= = [N {word = [Z:men]}]}]
```

とすれば、(b1)σ_1 = (b2), (b2)σ_2 = (b3), (b3)σ_3 = (b4) である。
(a1)$\theta_1\theta_2$ = (a3) = (b4) = (b1)$\sigma_1\sigma_2\sigma_3$ なので、(a1) と (b1) は単一化し、unifier は、$(\theta_1\theta_2, \sigma_1\sigma_2\sigma_3)$ である。

4 TUG に対応するプログラム

TUG に対応する GLP プログラムの例を示そう。まず、APPEND のプログラムは、

```
[APPEND {f = [NIL], b = [X:LIST], w = [X:LIST]}] ←.

[APPEND {f = [CONS {first = [A:ALL], rest = [X:LIST]}],
         b = [Y:LIST],
         w = [CONS {first = [A:ALL], rest = [Z:LIST]}]}]
← [APPEND {f = [X:LIST], b = [Y:LIST], w = [Z:LIST]}].
```

となる。文や名詞句に関するプログラム (の一部) は、次のようになる。

```
[sentence {string = [Z:WORD],
           c_str = [S {np = [N:ALL], vp = [V:ALL]}]}]
← [noun_ph {string = [X:WORD], c_str = [N:ALL]}],
  [verb_ph {string = [Y:WORD], c_str = [V:ALL]}],
  [APPEND {f = [X:WORD], b = [Y:WORD], w = [Z:WORD]}].

[noun_ph1 {string = [Z:WORD],
           c_str = [NP1 {det = [D:ALL], noun = [N:ALL]}]}]
← [det {string = [X:WORD], c_str = [D:ALL]}],
  [noun {string = [Y:WORD], c_str = [N:ALL]}],
  [APPEND {f = [X:WORD], b = [Y:WORD], w = [Z:WORD]}].
```

辞書項目を記述するために、節：

```
[LEX {c_str = [X:LCAT]}] ← [X:LCAT].
```

があり、これによって、あとは各単語ごとの記述をする節があればよい。3例のみ示す。

```
[DET {word = [all], agr = [pl]}] ←.
[LIKE {word = [likes], agr = [sg]}] ←.
[MAN {word = [men], agr = [pl]}] ←.
```

5 Resolution による構文解析と文生成

5.1 Resolution による構文解析

文 <Mary likes all men> を構文解析して、その意味表現を得るためには、上記のプログラムに対して、

```
[sentence {string = <Mary likes all men>}]
```

という goal を与えればよい。Prolog と同様の SLD Resolution によって、

```
[sentence
  {string = <Mary likes all men>,
   c_str = [S {np = [NP2 {name = [PN {word = [Mary]}], agr = [sg]}],
               vp = [VP {v = [LIKE {word = [likes], agr = [sg]}],
                         np = [NP1 {det = [DET {word = [all], agr = [pl]}],
                                    noun = [MAN {word = [men], agr = [pl]}],
                                    agr = [pl]}],
                         agr = [sg]}]}]}]
```

が得られる。この解析結果は、[6] に書かれた TUG としての解析結果と対応している。

5.2 Resolution による文の生成

逆に、意味表現から、それに対応する文を生成するには、上記のプログラムに対して、たとえば、

```
[sentence
  {c_str = [S {np = [NP2 {name = [PN {word = [Mary]}], agr = [sg]}],
               vp = [VP {v = [LIKE {word = [likes], agr = [sg]}],
                         np = [NP1 {det = [DET {word = [all], agr = [pl]}],
                                    noun = [MAN {word = [men], agr = [pl]}],
                                    agr = [pl]}],
                         agr = [sg]}]}]}]
```

という goal を与えればよい。SLD Resolution によって、構文解析の場合とまったく同じ結果が得られるので、意味表現に対応する文が、<Mary likes all men> であることがわかる。この文生成の結果は、[6] に書かれた TUG としての文生成の結果と対応している。

6 おわりに

本論文では、TUG を GLP のサブクラスとみなすための方法を与えた。これによって、TUG の宣言的意味論や SLD Resolution や プログラム変換などの厳密な理論的基礎が、GLP の理論から自動的に得られたことになる。GLP の理論を体現するプログラミング言語 UL/α を使えば、TUG の処理系の作成は容易である。本論文の実行例はそのようにして作成されたシステムを用いて実際に動かした結果である。今後の課題は、

- UL/α コンパイラ [5] などを使って、TUG の高速な処理系を得る技術を開発すること
- GLP と UL/α の枠組を用いて、TUG の枠組をさらに拡張すること
- TUG のプログラム変換によって、高度な自然言語処理システムを構築すること [7]

などがある。

References

[1] Akama,K. : Declarative Semantics of Logic Programs on Parameterized Representation Systems, *Hokkaido University Information Engineering Technical Report*, HIER-LI-9001 (1990)

[2] Akama,K. : Generalized Logic Programs on Specialization Systems, *Preprints Work. Gr.for Programming, IPSJ* 6-4-PRG, pp.31-40 (1992)

[3] 赤間 清 : 一般化論理プログラムの Unfold 変換, 関数プログラミング JSSST'91, 近代科学社 (1992)

[4] Carpenter,B. : The Logic of Typed Feature Structures, Cambridge Tracts in Theoretical Computer Science 32, Cambridge University Press (1992)

[5] 出葉 義治, 繁田 良則, 赤間 清, 宮本 衛市 : 一般化論理プログラミング言語 UL/α のコンパイラ, 情報処理学会, プログラミング研究会, 9-7-PRG, pp.49-56 (1992)

[6] Emele,M. and Zajac,R. : Typed Unification Grammars, Proc. of COLING '90 pp.293-298 (1990)

[7] 野村祐士, 赤間清, 宮本衛市 : プログラム変換による意味の解釈, 情報処理学会, 自然言語処理研究会, 本号 (1992)

[8] Shieber,S.M. : An Introduction to Unification-based Approaches to Grammar, CSLI Lecture Notes Number 4, Stanford University (1986)