ICOT Technical Memorandom: TM-1113

TM-1113

演繹オブジェクト指向データベースについて

横田　一正

September, 1991

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191 ~5
Telex ICOT J32964

Institute for New Generation Computer Technology

演繹オブジェクト指向データベースについて*

横田一正
(財) 新世代コンピュータ技術開発機構

1991 年 9 月 24 日

1 はじめに

『演繹オブジェクト指向データベース』(Deductive Object-Oriented Databases; DOOD) という言葉については皆さん大多数の方、聞かれたことがあると思いますが、その内容についてご存知の方は非常に少ないと思います。したがって、今日の話はこの内容について理解して頂くことに重点を置きます。この DOOD にこれまで注目していた方も、まずこのタイトルを見て意外に思われたかも知れません。なぜかというのは 3.1 節で説明します。さて話の構成ですが、まず 2 節で、動機となったいくつかの例を説明します。次に 3 節で、DOOD とは何かについて一般的に話したいと思います。後半の 4 節では、それをいかに実現するかについて、いくつかの理論的あるいは技術的なポイントをお話します。

2 例

DOOD は従来のデータベースでは対処が難しかった応用を考慮しています。いくつかの例を本節では紹介しましょう。

1) 関係データベースが問題とされた有名な応用に CAD があります。この応用では、下記 3) の複合オブジェクトの他にオブジェクト共有 (シェアリング) という考えが重要です。たとえば図-1 で、P_1, P_2, P_3 はそれぞれオブジェクトです。P_1 と P_2 が、P_3 を共有しています。つまり P_3 が何らかの理由で更新された

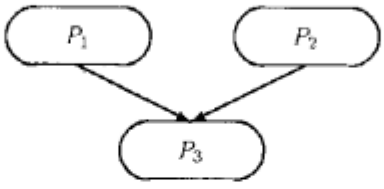

図 1: オブジェクト共有

場合、当然 P_1, P_2 から見える P_3 は共に更新されます。このような共有をいかに表現し実現するかが重要です。もちろん、Prolog で実現できないことではありませんが、オブジェクト共有を、応用 (ユーザ・プログラム) の意味論に依存した形ではなく、システムとしてどうサポートするかというのがここでは重要です。

*本稿は、Logic Programming Conference'91 (1991 年 8 ～ 11 日、東京機械振興会館) の招待講演に加筆修正を行ったものである。

2) 次に無限構造を考えましょう。知識表現ではよく出てくる構造です。図-2 はその簡単な例です。具体例

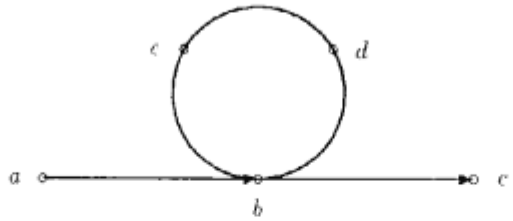

図 2: 巡回路を持ったパス

としてはいろいろなものを考えることができます。この例では、山手線と京浜東北線を考えることもできます。このような巡回構造を含むグラフ上でパスを、たとえば a から c のパスを考えましょう。b, d, c の巡回路は何回、回ってもかまわないので、無限個のパスが考えられます。もし個々のパスをオブジェクトと考えるならば、無限のパス、つまり無限構造をいかに表現するかが問題になります。通常パスはアークから再帰的に定義されるので、ルールによって定義される内包的なオブジェクトをいかに表現しオブジェクト識別子をいかに生成するかという問題とも考えることもできます。

3) Prolog ではいくつかの引数によってさまざまな対象を表現しますが、一般的にはもっと複雑な構造を表現したいことがあります。そのために項表現が問題になります。たとえば図-3 は、いわゆる複合オブジェクトの例です。これは“アキラ”という一つのオブジェクトを表しています。“アキラ”はいくつかのプロパ

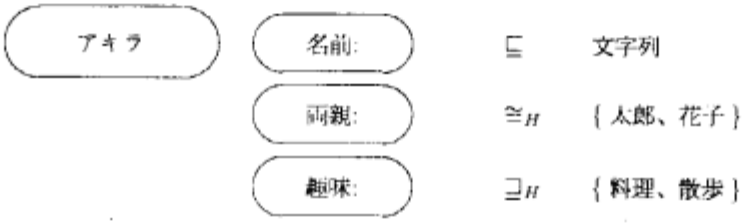

図 3: 複雑なプロパティ

ティ（属性あるいは性質、スロット）を持っています。たとえば“名前”とか、“両親”とか、“趣味”です。まず“名前”ですが、具体的な値が不明で、単に“文字列”であるという型しか分かっていないとします。次に“両親”については、まさに二人、“太郎”と“花子”であると分かっています。そして“趣味”ですが、少なくても“料理”と“散歩”の二つは持っています。これを表現したのが図-3 です。これらは包摂関係 $\sqsubseteq$ とそれを集合に拡張した Hoare 順序 $\sqsubseteq_H$ で表現できます (図の $\cong_H$ は $\sqsubseteq_H \wedge \sqsupseteq_H$ を表しています)。複雑なプロパティは、この例以外でもいくらでも考えられますが、それらをいかに統一的に扱うかが問題です。

4) 次の例は矛盾の扱いです。大規模知識ベースを構築する際には、雑多な知識を格納しなければならないので、それらの間の無矛盾性が大きな問題点になります。図-4 はその例です。P が 1 であるという情報と P が 2 であるという情報を一緒にすると矛盾してしまいます。しかし、たとえば A さんの意見では P は 1 であるが B さんの意見では P は 2 であるという形で、分類できればそれを避けることができます。一般にはもっと複雑で雑多な知識があるので、これを分類するためにはどんな概念を導入すべきかが重要です。この例では、枠で分類の一つの単位を表し、矢印で継承を表しています。ただしこの矢印は通常の継承とは逆です。これによってこの場合、図-4 の一番下の枠以外の P は矛盾を生じません。つまり矛盾を局所的に押し込めているのです。このような分類機能が必要とされます。

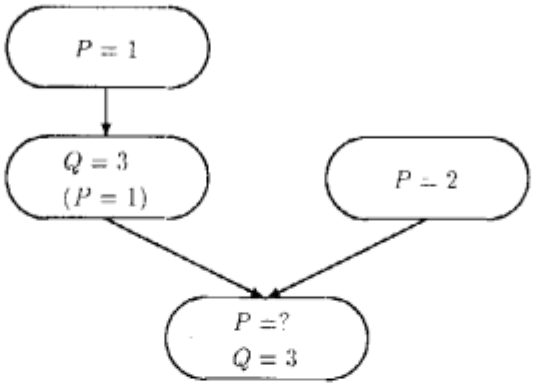

図 4: 矛盾の扱い

5) 少々生々しい例を見てみましょう [2]。Quintus Prolog で作成している遺伝子データベースの例です。図-5 はその文献情報に当たっています。object が述語記号で、引数は三つだけです。注目すべきなのは第

```
object(ref('Patterson et al (1981)'),
       1991/4/24,
       [kind(paper),
        authors(['D. Patterson',
                 'S. Graw',
                 'C. Jones'
                ]),
        title('Demonstration by somatic cell genetics of coordinate
             regulation of genes for two enzymes of puring synthesis
             assigned to human chromosome 21'),
        journal('Proc. Natl. Acad. Sci. USA'),
        volume(78),
        pages(405-409),
        year(1981)
       ]).
```

図 5: 文献情報

3引数です。文献情報の特徴として属性の数が可変であるために、リストの中にタームの形の属性 - 値対を入れています。その中で著者 (authors) は複数の場合があるのでその値も同じリストで表しています。通常の引数としてそれら属性を表現するには実用上困難なために、このような表現になっています。ところがここで問題なのは、組構成子と集合構成子をリスト構成子でオーバロードしていること、また通常のリスト単一化では比較ができないことです。つまりこれらは、通常の述語表現ではレベルが低過ぎてユーザ・プログラムの意味論に依存しています。このようなデータは数多くの応用に見ることができます。したがって先ほどの複合オブジェクトと同様、いかにこれを自然に表現し、その意味をサポートするかが問題です。

6) 図-6 も同じような例です [2]。会社 (company) がリストのリスト構造で表現されています。先ほどの著者と同様の問題がここにはあります。

```
object(restriction_enzyme('NotI'),
        1991/4/30,
        [prototype(none),
         recognition_sequence("GCGGCCGC"),
         cutting_site(2),
         sources([company('Amersham'),        company('BRL'),
                  company('Stratagene'),      company('Palliard Chemical'),
                  company('ESP Fermentas'),   company('BioExcellence (formerly Anglian)'),
                  company('IBI'),             company('Janssen Biochimica'),
                  company('Takara'),          company('Northumbria Biologicals Ltd.'),
                  company('USB'),             company('New England Biolabs'),
                  company('Toyobo'),          company('PL-Pharmacia-LKB'),
                  company('Promega Biotec'),  company('Molecular Biology Resources'),
                  company('Sigma'),           company('Boehringer Mannheim'),
                  company('Serva'),           company('New York Biolabs')
                 ]),
         references(['REBASE ref'(36),
                     'REBASE_ref'(477)])
        ]).
```

図 6: 遺伝子データ

データベースの分野では、CAD や CASE、OA、マルチメディアなどの、いわゆる「新しい」応用に対処するために、関係データベースの拡張として数多くのデータモデルが提案されてきました。論理プログラミングと密接な関係がある（ときにはどこが違うのかという質問さえ出される）演繹データベースは、問合せ最適化とか、意味論の問題が多く研究されてきましたが、このような様々な現実の応用に直面しますと、無力感さえ感じてしまいます。これらが演繹データベースを DOOD に拡張しようという大きな動機になっています。また DOOD が数多くのデータモデルの拡張になっていることも、この後の話で理解して頂けることと思います。

3 演繹オブジェクト指向データベースとは何か

この節では、演繹オブジェクト指向データベース (DOOD) の経緯からいくつかの周辺の話題も関連させて、DOOD とは何かを考えてみましょう。

3.1 経緯

DOOD の意図を一言でいうと、演繹データベースとオブジェクト指向データベースの統合です。当初は技術的な見通しが曖昧なこともあったために、「融合」という言葉を使っていたのですが、いまや「統合」とはっきりいってもかまわないでしょう。言い換えると、論理とオブジェクト指向概念のデータベースの視点からの統合といってもよいでしょう。

まず名称について説明しましょう。deductive databases (DDB) と object-oriented databases (OODB) の統合ということで deductive and object-oriented databases (DOOD) というのはごく自然に思えます。ところが実際には deductive と object-oriented のどちらを前に出すかで、随分揉めたこともあります。略号についても同様です。"O" を一つ取ると ("DOD") キナ臭くなりますし、その状態で "O" と "D" を逆にすると奇妙に ("ODD") になってしまうので、これも悩ましいものがありました。結局名称が安定するまで半年くらいかかったと記憶しています。日本語では「統合」を強調するために「演繹＋オブジェクト指向データベース」と "+" を使っていた時期もあるのですが、これは不自然だということで「演繹・オブジェクト指向データベース」と "・" になりました。ところが DOOD が実際に姿を現し始めると、deductive and object-oriented databases の "and" は英語の語としては不自然だということで、deductive object-oriented databases として使われ始めています。日本語としては "・" をはずして「演繹オブジェクト指向データベース」となります。これが本講演のタイトルについて最初に言及したことでした。数週間前に行われた International Conference on Logic Programming のプログラムを見ると、object-oriented deductive databases とか object-oriented logic programing という言葉も使われています。論理とオブジェクト指向概念の統合という視点から考えれば、ほぼ同じと考えてもよいと思います。

次に経緯について触れたいと思います。この用語の誕生には二つの契機がありました。一つは、ICOT の演繹・オブジェクト指向データベース・ワーキンググループで、1988 年の 4 月に発足しました[1]。もう一つは、1989 年 12 月に京都で開催された第 1 回演繹・オブジェクト指向データベース国際会議 (The First International Conference on Deductive and Object-Oriented Databases, DOOD89) です[2] [1]。DOOD89 の準備は会議の 1 年半以上前から始まっていました。この DOOD89 で "DOOD" という言葉が始めて一般的に知られるようになったといってよいでしょう。DOOD89 の基調講演は ICOT の渕所長で

"Towards a New Step of Logic Paradigm"

と題しています。これは DOOD がデータベースだけでなく今後の論理プログラミング言語全般に大きな影響を与えると予想されたからです。現状はこの予想通りに進んでいます。DOOD 国際会議は 2 年毎に開催されることになっており、第 2 回は今年 12 月にミュンヘンで開催されます。第 3 回は 1993 年ですが、アメリカか日本での開催を予定しています。

以上簡単に DOOD の経緯を眺めてきましたが、それではなぜ DOOD が必要になってきたのかを、別の観点から考えてみたいと思います。

3.2 関係データベースの拡張

1990 年の 3 月に "Extending Database Technology" という国際会議が開催され、非常におもしろいパネルが行われました。そのタイトルは

> Why are *Object-Oriented* Folks Producing *Systems*, while *Deductive* Folks are Producing *Papers*?

[1] この経緯としては、1987 年の 11 月始めのヨーロッパ出張、その帰国後の同月末に京都で行われたソフトウェア科学会全国大会、などでの議論から演繹データベースとオブジェクト指向データベースとの統合を次世代データベースの中心として考えようとして、何人かの方との議論のを経て、ICOT の DOOD のワーキンググループを作ったのです。

[2] DOOD ワーキンググループのメンバーでもある大阪大学の西尾先生は、データベース理論のワークショップを考えていらっしゃって、やはり演繹データベースとオブジェクト指向データベースの統合という観点から、DOOD89 を組織されたのです。

というものです。これはいかにも現状を表しており、最初見たときは思わずほくそえんでしまうほどでした。つまりオブジェクト指向データベースは応用指向で、理論がなくてもユーザの責任でより多くの応用に対処できれば良い、という傾向が非常に強いのです。何がオブジェクト指向データベースかというコンセンサスがなくても 10 指に余るデータベース管理システムが商品としてできてきている状態です。一方、演繹データベースは、問合せ最適化や意味論などの基礎研究に偏重してきた感があり、どちらかというと応用を軽視してきたといえると思います。もちろん演繹データベースの研究者は無限の応用の可能性を信じてはいるのですが、2 節の例を見るとそのままではかなりの困難を感じます。この大きなギャップをいかに埋めるかが、DOOD の意図でもあります。

関係データベースから演繹オブジェクト指向データベースまでの流れを、非常に荒らっぽいのですが図-7 に示します。関係データベースが最初に提案されたのが 1970 年で、研究面では 70 年代というのは関係データベースの時代であったといっても過言ではないと思います (開発面では後ろに 5 年くらいシフトしていますが)。しかしよく知られているように関係データベースはすべてのデータを正規形と呼ぶ平坦な形にしてしまいます。したがって単純な応用の場合は良いですが、現実には対処するのが難しい応用が非常に多くあります。問題は表現や意味だけに止まらず、正規化に伴う関係の分割によって、悪名高い結合演算の増加で処理効率も良くないわけです。そこでさまざまな拡張が提案されてきています。たとえば 1976 年に実体関連モデル、1977 年に非正規関係モデルと演繹データベース、1982 年に複合オブジェクトモデル、1985 年にオブジェクト指向データベース、という具合です。そして DOOD は 1989 年ということになり、図-7 の位置を占めています。

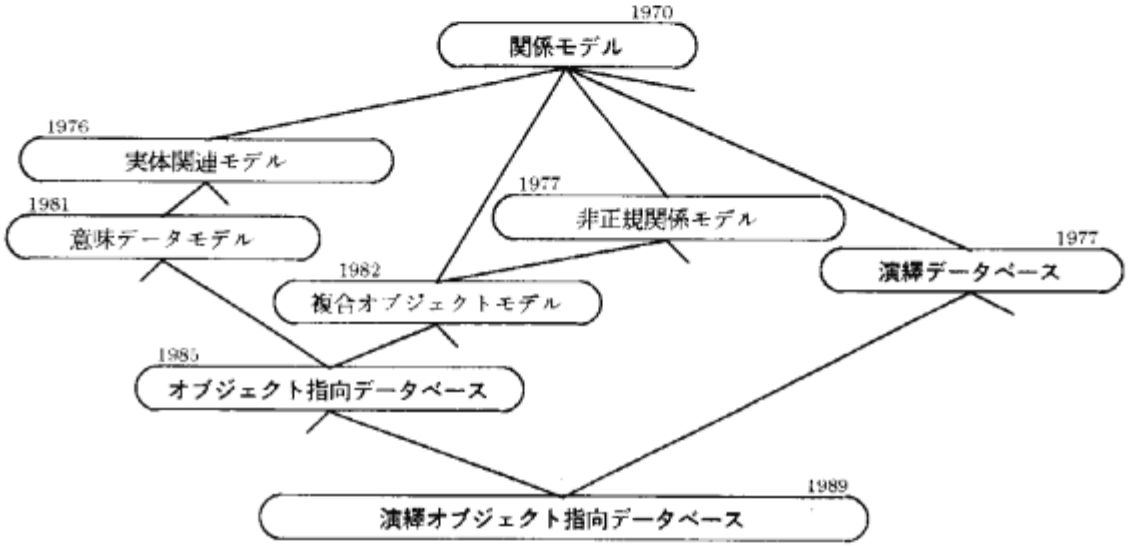

図 7: 関係データベースの拡張

演繹データベースとオブジェクト指向データベースの統合を考えるためには、両者の間のギャップについてもう少し考えてみたほうがよいでしょう。まず第一にこのギャップの大きさをいかに評価するかです。研究対象として考えるとき、それは大き過ぎても小さ過ぎても駄目なわけです。この DOOD は、結果的にほどよい研究対象となったと考えられます。次に考えなくてはならないのは、オブジェクト指向データベースと演繹データベースの間のギャップと、オブジェクト指向プログラミング言語と論理プログラミング言語の間のギャップは同じか、という問題があります。これはまたプログラミング言語とデータベースの違いにも関連します。この点を 3.3 節で少し考えてみましょう。

3.3 データベースとプログラミング言語

最近データベースとプログラミング言語の関係が盛んに議論されており、両者の統合がデータベース・プログラミング言語 (DBPL) として盛んに研究されるようになっています [5]。ここではその中でも論理パラダイムに焦点を当てた DBPL を考えてみましょう。

両者の統合は主にインピーダンス・ミスマッチの解消という視点から研究されています。プログラミング言語に永続性を導入したものを永続プログラミング言語 (PPL) といいますが、DBPL には、プログラミング言語にデータベース言語の機能をもたせたアプローチと、データベース言語のプログラミング能力を計算完備にすることを目指したアプローチの二つがあります。この意味で DBPL は PPL を含んでいるといっても良いでしょう。データベース言語にはデータ操作言語だけではなくデータ定義言語がありますから、この統合には型の概念が重要な役割を果たします。演繹データベースは論理プログラミング言語としてのプログラミング能力を備えていますが、"データベース" という名前とは裏腹にデータベース機能を考慮していなかった、といえるのではないでしょうか。ほとんどはファクトの集合である外延データベースを既存の関係データベースと対応させるだけで、永続性を正面から扱ってこなかった、といえるでしょう。最近議論されているのは、DBPL の "DB" とはデータベース機能のどこまでを含んでいるのかということです。たとえば同時実行制御や二次記憶管理などのいわゆる管理機能まで DBPL に持たせるべきかどうか議論があります。DOOD は演繹データベースの拡張でもありますので、プログラミング言語の能力は持っています。したがってこの問題は DOOD を考える上でも議論すべき大きな問題です。

オブジェクト指向の観点からは、オブジェクト指向プログラミング言語とオブジェクト指向データベースがあるわけですが、オブジェクトがもともと手続きを含んでいるため、DBPL の意識は当初から強くありました。これらに永続性と論理パラダイムを追加すれば、言葉としては、論理型 OODBPL と DOOD にそれぞれなりますが、両者は同じものと考えてもかまわないと思います。これが異なった視点から見た DOOD です。

しかし論理型 OODBPL という場合、疑問が出てきます。論理型とオブジェクト指向型がそれぞれ何を意味しているのかという問題です。これには二つの視点が必要です。まずデータモデルの観点からは、対象とするのがオブジェクトなのか論理的要素なのかということです。何をオブジェクトの粒度にするかについては後で触れますが、たとえばオブジェクトの定義を論理プログラミング言語における述語定義と同じにみなすこともできます。述語定義は関係モデルにおけるスキーマ設計に対応していますが、データベースは元来実世界のオブジェクトをいかに自然に計算機の中に表現するかを出発点にしていますので、拡張した述語定義をオブジェクト設計に対応させるのは非常に自然に思えます。次に計算モデルの観点からは、たとえば Prolog に対して手続き的意味論として SLDT や項書換えの意味論を与えることができるように、論理型、オブジェクト指向型双方の意味論を与えることができることが知られています。統合の元になっている言語の計算パラダイムは結果の DBPL の計算パラダイムの一つとして引き継がれます。図-8 はそれらの関係を表しています。一昨年の DOOD 国際会議で "Object-Orientes Database System Manifesto" が発表されました。これによるとオブジェクト指向データベースが持つべき性質を、mandatory features、optional features、open choices の三つに分類していいます。この中で面白いのは計算パラダイムというのが open choices に入れられていることです。私自身はこの Manifesto には必ずしも賛成できないのですが、この点については同意見です。もっともオブジェクト指向がその中の一つでなければならないのは当然ですが。これらが上で述べた、論理型とオブジェクト指向型の二つの形容の意味です。

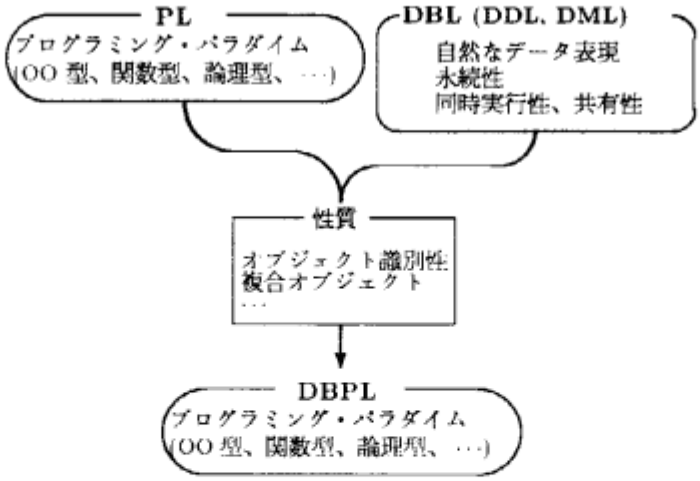

図 8: OODBPL の枠組

3.4 DOOD の枠組 [3, 4]

さて DOOD は論理とオブジェクト指向を元にした新しいデータベースのパラダイムですが、アプローチとしては、オブジェクト指向データベースの拡張として考えるよりは演繹データベースの拡張として考える方が妥当だと考えています。というのはオブジェクト指向データベースの場合にはいまだに理論的な基盤が明確ではありませんが、演繹データベースの場合には述語論理というはっきりした基盤があるからです。したがって DOOD へのアプローチとして、演繹データベースを拡張するという方向で考えてみましょう。

そのアプローチは、データベース機能の導入を除いて、以下のように分類することができます [3, 4]。

1) 論理的拡張

   (a) 否定、選言、限量子など不確定情報の導入

   (b) 集約関数、確定因子などの導入

2) データモデル的拡張

   (a) オブジェクト識別性の追加、オブジェクト共有の実現

   (b) 非正規関係や複合オブジェクトなどの複雑なデータ構造の組込み

   (c) データと手続きを一体化したカプセル化

3) 計算モデル的拡張

   (a) メッセージパッシング (オブジェクト指向計算)

   (b) 制約解消系の追加 (制約論理型スキーム)

これら三つの拡張の方向は必ずしも直交したものではありません。

たとえば次の例を考えて下さい。

$$p(X, <Y>) :- q(X, Y)$$
$$r :- \cdots, p(X, \{\}), \cdots.$$

$?- r.$

最初のルールの "< ··· >" は集合のグルーピングをおこなうための構成子で、MCC の $\mathcal{LDL}$ で採用されています。ここではある X に対して $q(X,Y)$ を満足するすべての Y の集合を $<Y>$ に求めることを意味しています。これを導入すると、2番目のルールのように空集合を書くことができます。つまり集合のグルーピング結果が空集合になるということは、それを満足するものが存在しないことですから、negation_as_failure の意味での否定を表現することができます。

また次の例を考えて下さい。

$$S \sqsubseteq_H \{a,b\} \Leftrightarrow S \sqsubseteq_H \{a\} \vee S \sqsubseteq_H \{b\}$$
$$p \leftarrow o/[l \rightarrow .\{a,b\}. \qquad (\Leftrightarrow o[\{o.l \sqsubseteq_H \{a,b\}\}.)$$
$$?- p/[l \rightarrow a].$$

要素間に順序を導入すると、その集合間の順序は通常の包含関係とは異なったものになります。この例で $\sqsubseteq_H$ は要素間の順序を元にした Hoare 順序を表しています。すると、ここにあるように選言が自然と導入されることになります。二つ目はそれがルールの中に制約 ($\|$ の右) として出現した例です。

これらの例は、DOOD のための上で示した拡張の 1) と 2) が必ずしも直交しないことを表しています。その他、2) と 3) については、たとえばメソッドが定義されなければメッシージ・パッシングも定義できないので、それらが直交しないことは明らかでしょう。したがって演繹データベースの拡張を考えるときは、これら三つの方向の拡張がどのように関連しているかを睨む必要があります。

4 DOOD の技術的課題

さて DOOD についてのイメージは湧いてきたでしょうか。この節では、DOOD に関してのいくつかのトピックをお話ししましょう。

4.1 オブジェクトとは何か?

これまでオブジェクトという言葉をずっと使ってきましたが、オブジェクトとは何かを少し考えてみましょう。論理プログラミング言語で問題になるのは、内包オブジェクトあるいは仮想オブジェクトの扱いです。たとえば次の $path$ のプログラムを考えてみましょう。

$$path(X,Y) :- arc(X,Y).$$
$$path(X,Y) :- arc(X,Z), path(Z,Y).$$

もし $path$ をオブジェクトと考えるならば、このプログラムは、上記のルールによって生成される $path$ という内包的オブジェクトを定義しています。しかしこの定義はパスの起点と終点しか定義していないので、図-9にある $a \rightarrow b$ と $a \rightarrow c \rightarrow b$ の二つのパスを別のオブジェクトとして考えることができません。それを区別するためには以下のようにプログラムを書き換える必要があります。

$$path([X,Y],X,Y) :- arc(X,Y).$$
$$path([X|W],X,Y) :- arc(X,Z), path(W,Z,Y).$$

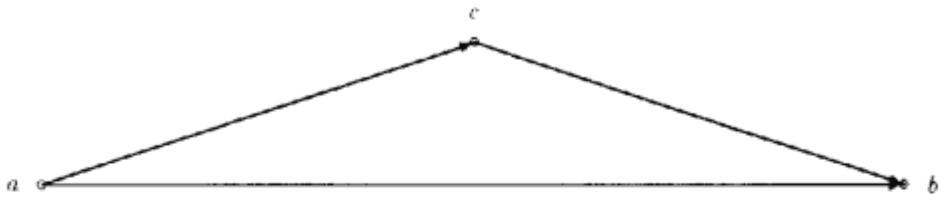

図 9: 複数のパス

つまり *path* の第 1 引数に経路情報を持たせ、それらを区別するようにしたのです。しかし巡回路を持った図-10 のパスは、このプログラムでは表現できません。

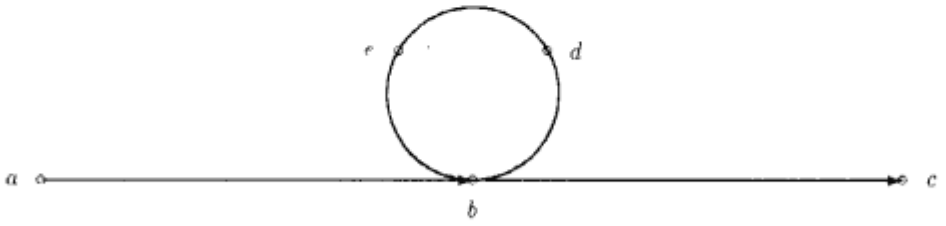

図 10: 巡回路を持ったパス

これらの例に見られるように、何をオブジェクトにするかは応用に依存しています。そしてそのニーズに合った表現能力が言語に要求されるのです。

次にオブジェクトの定義について考えてみましょう。論理プログラミング言語でのオブジェクトの定義は二つに分類できると思います。

1) まず、一つの項表現によってオブジェクトを表現する方法です。各オブジェクトはオブジェクト識別子によって区別されますが、この場合は同一の識別子をもった項表現の集まりによって一つのオブジェクトが表現されます。論理プログラミング言語の場合ルールもあるので、ヘッドに同一になる可能性を持った (つまり単一化可能な) オブジェクト識別子をもったルールの集合としてオブジェクトを定義するといったほうが、より正確でしょう。Prolog のタームがオブジェクト識別子であるとすれば、オブジェクト定義は述語定義に対応すると言っても良いかも知れません。先程の *path* がその例です。一つのオブジェクト定義は内包的なオブジェクトを定義しているだけですから、これから複数の、場合によっては無限個の具象オブジェクトが生成されます。このように項表現によってオブジェクトを表現する方式を使っている言語としては、IQL、ILOG、F-logic、*Quixote* などがあります。

2) もう一つは、ルールの集合をオブジェクトとして定義する方法です。1) と違う点は、1) の内包的オブジェクトが単一化可能なオブジェクト識別子をヘッドに持つルールの集合であったのに対し、ここでのルールの集合には一般には制限はありません。そしてこの集合に対してオブジェクト識別子が付けられるのです。これは、上記の定義に比べると一般的には粒度の大きなオブジェクトとなりますが、知識を整理するのに便利です。たとえば、本の構成を考えれば章節により分類されていますし、ソフトウェア工学的にも実用的です。この方式を採用している言語に、ESP や *Quixote* があります。もちろんここでは非常に単純化して考えているわけで、実際には ESP がそうであるように、これ以外にさまざまな仕掛けがあります。

これらは共に有用なものなので、二つをいかに統合するかが今後重要になってくるでしょう。とくにオブジェクト間の継承を考えると、両者をいかに調和させるかが技術的な問題になります。この二つを統合した言語

としては *Quixote* があります。

4.2 項表現 — 拡張項、部分項

それでは、オブジェクトをいかに表現すべきでしょうか。オブジェクトはオブジェクト識別子とプロパティ、メソッドとその実装から構成される、と簡単に考えることができます。ここではまず、それら表現の出発点となる項表現を考えてみましょう。一階述語論理の述語表現をどうするかが最初に問題になります。引数の個数が固定であること、引数の位置が固定であること、など述語表現の欠点は良く知られていますが、これは本質的な欠点なのか、大きなオブジェクトを表現する際の単に使いにくさなのか、を十分に検討する必要があります。

述語表現は項構成子によって、属性 - 値対 (attribute-value pair) としてよく書き直されます。たとえば

$$p(t_1, \cdots, t_n) \Longrightarrow p[l_1 = t'_1, \cdots, l_n = t'_n].$$

のように再帰的に定義します。ここで $l_1, \cdots, l_n$ は新しく導入されたラベル (属性名) で、"[", "]" は組構成子です。t'_i は t_i の属性 - 値対表現です。ラベルは引数の役割を表しているので、その位置はどこにあってもかまいません。さらに引数の数ですが、述語表現では引数の内容を問わない (分からない) ときに匿名変数を使いますが、ラベルによってそのような場合は明記する必要がなくなりました。つまり引数の位置と個数の問題から解放されたわけです。

もし問題がこれだけなら、表現上の本質的な問題ではなく、上の式で $\Longleftarrow$ も成立するでしょう。そこで二つの問題を考えてみましょう。

- まず第一に、2 節の図 -5 と図 -6 の例をもう一度考えてみましょう。この例での問題は、引数の個数を「本質的に」決定できないことでした。このように本質的に部分情報を扱わねばならない場合、あるいはスキーマが不確定である場合、述語表現では、例のようにユーザがその意味論に責任を持たない限り部分情報の表現は不可能です。
- 第二に、無限構造をいかに表現するかという問題があります。図 -10 のようなオブジェクトについても、述語表現にタグを導入することを考えることはできますが、やはりユーザがその意味論に責任を持つ必要性が出てきます。
- 第三は、図 -3 のような複雑なデータ構造に関連しています。構成子をいくつ導入するとか、制約表現にするとか、いくつかの方法が考えられますが、これらも第一、二の場合と同じことになります。

つまり、述語表現は、多くの応用で必要とするオブジェクトの意味表現にはレベルが低過ぎるのです。組構成子や集合構成子による属性 値対の項表現は、述語表現を拡張したという視点からは拡張項と呼ばれ、部分情報の表現の視点からは部分項と呼ばれています。

参考として図 -5、図 -6 を複合オブジェクトとして書き直した例を図 -11、図 -12 に示します。

4.3 オブジェクト識別性

さて、オブジェクト指向の分脈でのオブジェクト識別性の重要性はいうまでもありません。ここで考えたいのは、機能と表現の二つについてです。

```
object[ref='Patterson et al (1981)',
       date=1991/4/24,
       kind=paper,
       authors={'D. Patterson', 'S. Graw', 'C. Jones'},
       title='Demonstration by somatic cell genetics of coordinate
              regulation of genes for two enzymes of puring synthesis
              assigned to human chromosome 21',
       journal='Proc. Natl. Acad. Sci. USA',
       volume=78,
       pages=405-409,
       year=1981
      ].
```

図 11: 文献情報

まず機能に関してですが、第一に 2 節で見たオブジェクト共有の問題があります。たとえば

$$o_1/[l_1 = o_3], o_2/[l_2 = o_3] \implies o_3 \text{ は } o_1 \text{ と } o_2 \text{ に共有されている}$$
$$o_3/[\cdots]$$

を考えて下さい。o_1 と o_2 は l_1, l_2 というそれぞれ異なったラベルで o_3 を共有しています。o_3 の更新はそれらに共に反映されます。第二の機能として不変性 (immutability) があります。直観的には、A さんという人 (オブジェクト) の年齢 (プロパティ) が 21 才から 22 才に変化したとしても、A さんであることには変わりないということです。したがって

$$o_1/[l_1 = a] \implies o_1/[l_1 = b]$$

のようにプロパティが更新されても、それらが同じオブジェクト識別子 o_1 を持っていれば、一つの同じオブジェクトに関する記述になっています。またそれらは

$$o_1/[l_1 = o_2], o_1/[l_1 = o_3, l_2 = o_4] \implies o_1/[l_1 = o_2 \mid o_3, l_2 = o_4]$$

のように、一つの記述に書き直すことも、あるいは逆に分解することも可能です。ここで $\mid$ は二つの値 (オブジェクト) のマージ操作を表しています。オブジェクト識別子が非常に強い概念を持っていることがおわかりでしょう。

Prolog の場合、引数のどこかが異なっていると、それは別のタームになります。つまりオブジェクト識別性の観点から考えると、そのターム自体がオブジェクト識別子の役割を持っており、Prolog は、プロパティの表現を持っていないと考えることができます。

次に考えなければならないのが、表現の問題です。これについてもさまざまな議論がありますが、大きく分けると二つの立場が考えられます。つまりオブジェクト識別子をユーザに意識させるかさせないかです。一つの代表として ILOG という言語があります。たとえば 2 節の $path$ を考えると、

$$path(*, X, Y) :- arc(X, Y).$$
$$path(*, X, Y) :- arc(X, Z), path(*, Z, Y).$$

```
object[restriction_enzyme='NotI',
       date=1991/4/30,
       prototype=none,
       recognition_sequence="GC'GGCCGC",
       cutting_site=2,
       sources=[company={ 'BRL',     'Amersham',          'New England Biolabs',
                          'Takara',  'ESP Fermentas',     'BioExcellence (formerly Anglian)',
                          'IBI',     'Stratagene',        'Boehringer Mannheim',
                          'USB',     'Palliard Chemical', 'Northumbria Biologicals Ltd.',
                          'Toyobo',  'Promega Biotec',    'PL-Pharmacia-LKB',
                          'Sigma',   'Janssen Biochimica', 'Molecular Biology Resources',
                          'Serva',   'New York Biolabs'
                        }]
       references={'REBASE_ref'(36), 'REBASE_ref'(477)}
      ].
```

図 12: 遺伝子データ

のように "+" によってオブジェクト識別子を表現します。実際には、ヘッドの述語の全引数 (この場合 X と Y) の Skolem 関数となります。書換えをおこなうと

$$path(skolem(X,Y),X,Y) :- arc(X,Y).$$
$$path(skolem(X,Y),X,Y) :- arc(X,Z), path(skolem(Z,Y),Z,Y).$$

となります。したがって二つのルールから同一のパスが生成されるとそれらは同一のオブジェクトとなります。しかし IQL という言語では、ルールが異なればたとえ引数が同一でも別のオブジェクトという立場を取っています。これらは言語設計で大きな決断を必要とする点です。

表現に関するもう一つの考え方は、ユーザが明示的にオブジェクト識別子を指定することです。たとえば *Quixote* がこれに当たります。たとえば

$$path[from = X, to = Y]/[fee = P] \Leftarrow arc[from = X, to = Y]/[fee = P].$$
$$path[from = X, to = Y]/[fee = P] \Leftarrow path[from = X, to = Z]/[fee = P1],$$
$$path[from = Z, to = Y]/[fee = P2] | \{P = P1 + P2\}.$$

は、arc ごとに通過料金が決まっていて、$path$ の料金はそれを単純加算したものとするプログラムです。$path[from = X, to = Y]$ や $arc[from = X, to = Y]$ がオブジェクト識別子です。何をオブジェクト識別子とするか、何を可変なプロパティとするかは、プログラムで明示的に指定します。

これら二つの表現法には一長一短があります。最初に言いましたが、何をオブジェクトにするかは応用によって異なっています。この点では本質的には両者は同じです。ただ 2 節で述べたような無限構造を扱うためには、前者の表現法では難しさがありますので、そのような扱う対象によってこれらが決まります。

4.4 オブジェクトのプラットフォーム

ここで複合オブジェクトについて再び考えてみましょう。オブジェクトの複合構造を表現するために、これまでの例では組構成子と集合構成子を使いました。さらにリスト構成子とか多くのものが考えられます。また無限構造を表現するためにはタグの導入が考えられますし、複雑な属性を表現するためには制約の導入も考えられます。さらに汎化関係のような順序の導入も考えられます。ここで問題です。つまりどこまで複雑にすれば良いのか、またその場合の意味論はどうなるのかということです。ここではそれらのプラットフォームあるいは意味論について考えることにします。

まず簡単な例として、組構成子を集合表現することを考えましょう。組と集合は

$$p[l_1 = t_1, \cdots, l_n = t_n] \quad \Leftrightarrow \quad (p, \{(l_1, t_1), \cdots, (l_n, t_n)\})$$
$$(e_1, \cdots, e_n) \quad \Leftrightarrow \quad \{e_1, \cdots, e_n\}$$

のように変換できます (集合は無変換です)。ここで (a, b) は属性 - 値対 (a と b の順序対)、つまり $\{\{a\}, \{a, b\}\}$ を表しています。リスト構成子を導入してもこのような変換ができるのは容易にわかるでしょう。ただ集合構成子自体の意味論については別途考えることが必要です。たとえば

$$p[l = \{e_1, \cdots, e_n\}] \quad \Leftrightarrow \quad \{p[l = e_1], \cdots, p[l = e_n]\}$$

というものも考えられます。つまり集合構成子は複数のオブジェクトの略記表現と考えるわけです。これは一般的なベキ集合を許してしまうと計算量が大きくなってしまうので、それを避けるという意味もあります。いずれにしてもこのような集合表現は有力なプラットフォームの一つです。

それでは無限構造はいかに扱うべきでしょうか。知識表現では

$$person[parent \rightarrow person[parent \rightarrow person[parent \rightarrow [\cdots]]]]$$

のような無限構造がよく現れますが、上記のような集合では不可能です。そこで超集合論を考えてみましょう。超集合論は、ZFC 集合論の基礎公理を反基礎公理 (Anti-Foundation Axiom; AFA) で置き換えたもので、$x = \{x\}$、つまり $\{\{\{\cdots\}\}\}$ のような集合が許される世界です。これを使えば

$$X \mathbin{@} person[parent = X] \qquad X \cong person[parent = X]$$
$$o[l = X \mathbin{@} o_1[l_1 = Y], l' = Y \mathbin{@} o_2[l_2 = X]] \qquad X \cong o_1[l_1 = Y]$$
$$Y \cong o_2[l_2 = X]$$
$$W \cong o[l = X, l' = Y]$$

のように、集合を連立方程式として表現することができます。この超集合論のうれしい点はそのような連立方程式に一意的な解の存在が保証されていることです。この超集合論もプラットフォームの有力な候補です。

このように複合オブジェクトを考えるときには、表現のプラットフォームとその意味論を考えて行くことが非常に重要なポイントとなります。

4.5 型と階層

データベースではスキーマによってデータ構造を定義しますが、これは一般的には型あるいはクラスに相当しています。複合オブジェクト、あるいはオブジェクトについても同様の概念の導入を考えることができ

ます。型とクラスの違いは、通常は、型はオブジェクトの静的なテンプレートとして、クラスは動的なインスタンスの生成あるいは格納のために使われますが、ここでは型についてだけ触れることにします。型の利点はいうまでもなく、型推論や型検査によって誤りを事前にチェックすることの他に、型階層によるプロパティ継承の導入による表現の効率化があります。

型を考える上での出発点は、型と実体の関係について、それらを区別する否かという議論でしょう。データベースのスキーマの場合でも、データベース全体のスキーマをディクショナリとして管理し、さらに分散の場合はドメイン全体のディクショナリ情報を管理する必要が出てきます。つまりスキーマ自体を実体と同様に扱えれば、通常の関係の枠組で扱えるのです。つまり型を実体の汎化概念としてとらえた場合、必ずしも別世界として扱いたくない場合があるのです。別世界とした場合、実体間の汎化関係を考えるとさらにやっかいな問題が生じます。複合オブジェクトを対象にした場合、通常実体間の汎化関係は考えないので、型概念を別世界で考えることが多かったのですが、オブジェクト指向概念の観点からは、F-logic や DOT や *Quixote* のように両者を区別しないという言語が多く現れています。ψ-項あたりがそれらに大きな影響を与えたように思われます。このあたりの判断は、整数のような“個体”を実体と考えるか型と考えるかの判断にも依存しています。個々の整数のそれぞれにもメソッドを持たせうるのでそれらの型として扱うというのも自然ですが、実世界の応用からは不自然との考えも十分に理解できることです。

さて次に型(あるいはオブジェクト)間の順序ですが、これによってオブジェクト間のプロパティの継承を表現できます。多重継承を処理するためには、プロパティの値(型)のマージという処理が必要になります。

$$o_1 \sqsubseteq o_2 \ \wedge\ o_1 \sqsubseteq o_3 \ \wedge\ o_2/[l \rightarrow a] \ \wedge\ o_3/[l \rightarrow b] \ \supset\ o_1/[l \rightarrow \mathrm{merge}(a,b)]$$

この例は、o_1 が o_2 のプロパティ $l \rightarrow a$ と o_3 の $l \rightarrow b$ を継承しており、o_1 で a と b のマージが起こっていることを示しています。もし型間の順序が束構造をなしていれば、それは束演算に対応して一意的に決めることができます。*Quixote* ではプロパティを制約表現しているので、通常のような上から下への継承だけではなく、下から上への継承も行うようになっています。

$$o_1 \sqsubseteq o_2 \ \wedge\ o_2/[l_1 \rightarrow a] \ \supset\ o_1/[l_1 \rightarrow a]$$
$$o_1 \sqsubseteq o_2 \ \wedge\ o_1/[l_2 \leftarrow b] \ \supset\ o_2/[l_2 \leftarrow b]$$

多重継承を優先度で処理する方式もありますが、論理プログラミング言語の場合、述語間、節間の順序には通常意味がないので、これは相性が良くないように思えます。

もう一つ継承で考えなければならないのが例外です。これは継承が順序にしたがって連続的に生じるとき、何が不連続性を惹起するかを決めることである、と一言でいえるでしょう。この定義にはいくつか考えられますが、継承で変更可能なプロパティと不変なものを区別するのが自然だと考えています。たとえば

$$o_1[l \leftarrow a] \sqsubseteq o_1 \ \wedge\ o_1 \sqsubseteq o_2 \ \wedge\ o_2/[l \rightarrow b] \ \supset\ o_1/[l \rightarrow b] \ \wedge\ o[l = a]/[l = a]$$

では、$l \rightarrow b$ は o には継承されるが、$o[l = a]$ では同じラベルのプロパティ $l = a$ が不変であるので $l \rightarrow b$ が継承されないこと、つまり例外が生じていることを表しています。

4.6 プロパティとメソッド

次の話題として、論理プログラミング言語でメソッドをいかに扱うかについて簡単に触れましょう。例として次の *QUIXOTE* のルールを考えて下さい。

$$taro/[tel[loc = X] = Y] \quad \Leftarrow \quad \text{メソッドの実装部}$$
$$taro/[tel[loc = X, time = Z] = Y] \quad \Leftarrow \quad \text{メソッドの実装部}$$

この最初のルールのヘッドは、$taro$ というオブジェクトに $tel[loc = X]$ というメッセージを送ったら Y という返事が返ってくると読むことができます。いかに Y を決定するかが、$\Leftarrow$ の右、つまりボディ部に書かれるわけで、これがメソッドの実装部に相当します。$tel[loc = X]$ は構造を持っていますが、これはプロパティのラベルです。下の例はそれを少し拡張したもので、$taro$ への電話が時間によって変わることを表しています。ラベル自体にも拡張項表現を使うと、このような引数の可変性を表現するのに有効です。

一般的にオブジェクトのプロパティはすべてこのようにメソッドと解釈することができます。構造を持たないラベルは引数をもたないメッセージですし、ボディがなければ実装部を必要としていないメソッドになります。この視点から手続き的意味論を考えることも容易でしょう。

しかし気をつけなければならない点があります。次の例を考えて下さい。

$$o[l = X]/[l_1 \rightarrow a] \Leftarrow X \text{ は } 2 \text{ の倍数}$$
$$o[l = X]/[l_2 \rightarrow b] \Leftarrow X \text{ は } 2 \text{ の倍数}$$
$$o[l = X]/[l_2 \rightarrow c] \Leftarrow X \text{ は } 5 \text{ の倍数}$$

このようなオブジェクト定義に対して $?- o[l = 30]/[l_1 = X, l_2 = Y]$ という質問をするとどんな答が返ってくるでしょうか。上記のルールはいずれも $o[l = 30]$ なるオブジェクトの定義を含意していますから、すべてのルールに l_1 と l_2 のメッセージが送られます。最初のルールから l_1 の答として $X =\!\!\rightarrow a\ (X \sqsubseteq a)$ が返ってきますが、l_2 については下の二つの答が返されますので、同一オブジェクト識別子の下のプロパティ (解) のマージが必要です。つまり $Y =\!\!\rightarrow b \downarrow c\ (Y \sqsubseteq b \downarrow c)$ となります。$X =\!\!\rightarrow b\ (X \sqsubseteq b)$ や $X =\!\!\rightarrow c\ (X \sqsubseteq c)$ はそれだけでは解ではないのです。この例に見られるように、一つのオブジェクトを定義しているルールが複数あると、答の返される可能性のあるルールをすべて評価して解をマージすることが必要となります。

4.7 モジュール

あと、お話しておいたほうが良いと思うのは、オブジェクトの粒度のところでお話したもう一つの定義についてです。知識がある言語によって表現されたとすると、それらをある基準で分類するのがふつうです。オブジェクトの汎化関係による分類もその一つですが、数学の自然数論とか集合論といった理論の分類や本の構成のような分類機構が欲しくなります。いわゆる知識のモジュール化です。もう一つは、大規模知識ベースは必ずしも均質な知識だけではなく、互いに矛盾しあった知識を含む場合があるということです。矛盾を持った知識に対しては paraconsistent logic などが提案されていますが、これは矛盾し合う知識をそのまま扱うのではなく、矛盾値をもった知識として処理してしまうので大規模知識ベースには必ずしも適しているとは思えません。そこで矛盾した知識の住み分けのためにもモジュール概念が重要といえるでしょう。

モジュール概念の導入はさらにいくつかの拡張を可能にします。まずモジュール間に順序関係を導入することによって、ルールの継承を可能にすることです。これはたとえば、本の章節間に前提知識の順序があっ

たり、数学で線形代数が自然数論の知識を前提にするようなものです。たとえば図 13 はある本[3]の章間の関係を示しています。この図で各章間の知識の継承関係の例を見ることができます。この継承での、多重継

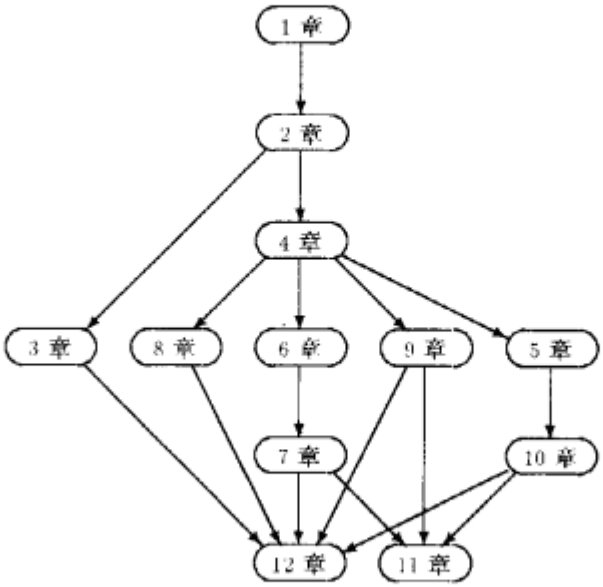

図 13: 本の章間の関係

承は論理プログラミング言語の場合は自然に導入できます。つまりルール集合の合併に過ぎないからです。例外については、たとえば数学の微積分の定義が教育段階によって変わっていくことを考えてください。古い定義は捨て去って、新しいものと置き換える必要があります。このような応用は多いのですが、合併と異なり、共通部分や差を求めることは一般的にはできないので、これには少し工夫が必要です。

次に仮説推論を考えることができます。モジュール間の順序関係を動的に変更することによって、仮説を付加した推論を行うことができますし、解の候補によってモジュールの枝を伸ばしてゆくことも可能でしょう。さらにモジュールを識別するためのモジュール識別子をパラメタライズすることによって、モジュールの抽象化を行い、実行時に必要に応じて具象化することもできるでしょう。もちろんこの他に、ソフトウェア工学的な視点からも、モジュール化が有用であるのはいうまでもありません。

4.8 問合せと解

問合せ処理については、解のマージについてこれまで何度か触れましたが、それ以外についてもう少し考えてみましょう。

まず制約との関係です。もちろん意味論に大きく依存するのですが、オブジェクトのプロパティは一般的に、オブジェクトとオブジェクト (の集合) 間のラベルを介した方向を持った関係と考えることができます。したがって、たとえば、

$$o/[l = o'] \iff o.l \cong o'$$

は o から o' への l を介した $\cong$ という関係で、ドット記法を導入することによって、制約として表現するこ

[3]田村『半群論』共立出版。

とができます。つまりオブジェクトを元にしたルールは、*Quixote* の例ですが

$$H \Leftarrow o/[l_1 \multimap a, l_2 = b] \iff H \Leftarrow o[\{o.l_1 \sqsubseteq a, o.l_2 \cong b\}$$

のようにボディ側のオブジェクトのプロパティを、制約論理プログラミング言語のように変形できます。したがってこの例の "$\{\cdots\}$" に対する制約処理系を考えることによって、制約論理プログラミング言語の一つとなります。

問合せは、単にゴールだけではなく、アドホックなルール集合 (データベース) を付加できるので、問合せ対象のデータベースに対して、ビューあるいは仮説を付加することができると考えられます。さらに複数のデータベース間の通信ができれば、ローカル・データベースとグローバル・データベースの区別や、永続ビューを考えることもできるでしょう。

次に解の概念ですがこれも変わってきます。一つはすでにお話した解のマージですが、もう一つは部分情報に関してです。プロパティの個数が可変であるというとは、指定されていないプロパティについての意味論が必要です。それを未定義と考えると、o という一つのファクトだけからなるデータベースに対して、$?- o/[l = a]$ という問いを出すと、単純に考えると、fail してしまうのですが、l のプロパティが未定義であることを考えれば、unknown あるいは not-yet-decided と答えるべきでしょう。あるいは「o の l の値が a に決まれば yes」というのが返されるべきでしょう。そこで条件付き解の必要性が出てきます。伝統的なデータベースの解は

$$?- Q.$$
$$DB \models Q\theta$$

の θ のような解代入として定義されました。一方、オブジェクト識別子を持つ言語の場合、

$$?- O/P.$$
$$DB \models O\theta/P_{O\theta}$$
$$DB \models O\theta/P_{O\theta}$$
$$\text{Answer} = \{O\theta/P_{O\theta}, \cdots\}$$

のように定義できます。ここで $P_{O\theta}$ はオブジェクト識別子 $O\theta$ によって決まるプロパティのマージを表しています。さらに、部分情報に関する条件の付加を考えると、一般的に

$$?- O/P.$$
$$DB \models O\theta/P_{O\theta}$$
$$DB, A \models O\theta/P_{O\theta},$$
$$\text{Answer} = \{(A, O\theta/P_{O\theta}), \cdots\}$$

のように、「DB に仮定 A を追加した $O\theta/P_{O\theta}$」という解が必要になってきます。もちろんこの条件は部分情報の扱いで必要になったので、それに対する制限を付加しないと解の意味がなくなってしまうでしょう。ここでは簡単に代入 θ を考えましたが、それが制約として一般化されるのはいうまでもありません。

4.9 永続性

次に重要な概念である永続性を考えてみましょう。永続性は、データベースにとっては本質的なのですが、通常の論理プログラミング言語にとってはメタな概念なので、これまであまり考慮されてこなかった、というのが実情です。

永続性については、もっとも素朴なものとして、プログラミング言語とデータベースのインタフェースをとるというものがあります。これが伝統的な両者の関係でした。この例として演繹データベースがあります。演繹データベースは、外延データベースを伝統的なデータベースに対応させ、内包データベースについては伝統的なデータベースに対応できないこともあって、扱いがはっきりしていません。そのために、問合せの一部、あるいは一時的なビューとして考えることもあります。次に、オブジェクトの観点からさらに永続性を進めると、オブジェクトを揮発オブジェクトと永続オブジェクトに二分する方法が現在一般的となっています。しかしここでの「永続性」は二次記憶に書くという物理的操作を意識したものであって、永続性の性質を十分に捕らえているとはいえません。そこでさらに一般化する必要があります。永続性はオブジェクトの生存期間に関する性質と定義したほうがすっきりするでしょう。その期間の具体例としては、トランザクション、プログラム、システムのシャットダウン、システム再生成などのタイミングや具体的な時間の指定が考えられるでしょう。

これら具体例でお分かりのように、永続性を言語だけでサポートするのは困難で、オペレーティング・システムを含めたシステム全体の問題と関係してきます。しかし、言語仕様に限れば、3 通りの方法があるのではないかと思っています。まず、オブジェクト (あるいは型) ごとに永続性を宣言する方法です。演繹データベースでの外延データベースの扱い方はこれに相当しています。もちろんこの中には暗示的な宣言というのも含まれています。次に、オブジェクトの特殊なプロパティとしてもたせることが考えられます。すべてのオブジェクトは生存期間に関するプロパティを持っており、これによってその期間が決定されるのです。最後の方法として、永続性を司る特殊なオブジェクトを想定することもできます。PCOB がこの方法を取っていますが、その特殊オブジェクトに生存期間を引数にしてオブジェクトを渡すと該当期間の生存が許されることになります。

もちろんこれら永続性を扱うためには、DBMS のような永続記憶管理機能が必要となりますので、それらとの連動したシステムが必要となります。

4.10 更新

永続性と関連して、更新の問題も重要です。オブジェクトの不変性の話もこの更新があって始めて意味をもちます。つまりプロパティが「更新」されても同じオブジェクトである、というのがその性質だからです。しかしこれは論理プログラミング言語における更新の意味論と関連しているので、理論的に検討すべき問題が数多くあります。

これら更新で重要なのは、更新の実行順序です。つまり更新の場合、実行順序の違いによって意味が異なってきます。たとえば $assert(a)$ と $retract(a)$ の実行順序を考えれば明らかでしょう。Prolog では AND- 並列と OR- 並列の 2 種類を扱いますが、この両者共に逐次化するための制御機構あるいは制限が必要になってきます。

次に重要なのがトランザクション概念です。つまり論理的な更新の単位です。たとえば

$$up(X) :- p(X); Y = X * 1.2; retract(X); assert(Y)$$

を考えて下さい。ここで "," の代わりに使っている ";" は逐次実行を示しています。このルールが何回か呼び出された後、何らかの理由で、このルールが実行中に fail してしまった場合、途中までの更新結果に意味があるかどうかという、という問題です。更新結果が永続性を持っていたとすると、プログラムを途中から開始するか、最初に戻した後に再実行しなければ、二重更新の危険性があります。前者がある種のチェックポイントで、後者がトランザクションですが、前者はこの場合データの共有を考えれば論理的に難点があるので、この例ではトランザクションが重要です。トランザクションは通常、開始宣言、正常終了宣言 (commit)、異常終了 (後戻し) 宣言 (rollback) の三つから構成されます。論理プログラミング言語での述語の呼び出しを考えれば、このトランザクションのネスト構造が必要になって来るように思えます。

一つのトランザクション内での更新の順序にも制限が必要でしょう。たとえば型階層やモジュール階層の更新があったとすれば、それらは継承関係に影響を及ぼしますので、継承前と後では同一ルールであっても結果が変わってしまいます。つまり、導出結果を保証するためには、ある種の更新は最上位のトランザクションの特定の (実行順序が保証されている) 場所でしか実行できません。したがってトランザクション論理の実行にはこのような保証も必要になります。

Prolog での assert や retract に相当する、オブジェクトの更新に関しては、意味論としては dynamic logic が簡単なのでよく用いられています。プロパティの更新に関しても同様ですが、同一オブジェクト識別子をもつオブジェクトがすでに存在していれば、マージが必要になるのはいうまでもありません。

このように更新については検討すべき課題が数多くあり、今後実用的な演繹データベースや DOOD を考えるとき、大きな問題となるでしょう。

4.11 関連の言語

さて最後に、DOOD に関連したあるいは DOOD を目指している言語を簡単に紹介しましょう。時間がなくなってきたので図-14 にその一覧を示すに留めます。 MCC で開発された $\mathcal{LDL}$ は集合のグルーピング機能を持った言語として提案されていましたが、今年の SIGMOD でオブジェクト指向概念を導入した $\mathcal{LDL}$++ として DOOD の仲間に新しく加わりました。その他についてはすでにいくつか論文が出ていますのでそれを参照して頂けたらと思います。

O-logic	Maier (1986); Kifer and Wu (1989)
F logic	Kifer and Lausen (1989)
DOT	Tsukamoto and Nishio (1989)
IQL	Abiteboul and Kanellakis (1990)
ILOG	Hull and Yoshikawa (1990)
Linear Object	Andreoli and Pareschi (1990)
Quixote	ICOT (1990)
$\mathcal{LDL}$++	Zaniolo et al (1991)

図 14: DOOD (関連の) 言語

5 おわりに

さて最後に、私自身の独断と偏見を交えて、まとめを行います。

DOOD は次世代データベースの最有力候補の一つであると考えています。古典的データベースはデータモデルとして内容が縛られることが多かったのですが、その拡張として提案されているものの多くは、その概念を越えています。DOOD はこれに対して、拡張の枠組を示したもので、多様なデータベースを統一的に考えようとするものです。またこれは、論理パラダイムの下で、データベース、プログラミング言語、人工知能の 3 者を統合しようという方向にも沿っています。

DOOD はこれからどうなるでしょうか。現在さまざまな言語が提案されていますが、現在まだ DOOD システムというのは存在していません。単に言語のレベルでの研究だけでなく、これらを DOOD システムとして実装し、システム面での研究を拡大して行く必要もあるでしょう。また演繹データベースの場合、応用作りに失敗したことは否めませんが、DOOD はどうなのか、今後多くの応用を通してその有効性を実証して行く必要があるでしょう。また今後重要になってくる技術として、人工知能との結合があります。知識の修正、知識の発見、知識の自己組織化など今後検討すべき点が数多く残っています。最後に触れておきたいのが DOOD の真ん中の "OO" についてです。これはオブジェクト指向を意味していますが、「いまさらなぜオブジェクト指向なのか?」という疑問があるかもしれません。最近ではエージェント指向とか新しい言葉も登場しています。ここで注意して頂きたいのは、DOOD の "OO" は必ずしも伝統的な "OO" に固執していないという点です。オブジェクト指向は必要な概念ですので、この "OO" の理論が、エージェント指向も含む今後の新しいパラダイムの基礎にもなると考えています。

謝辞

本講演の内容について、ICOT の DOOD ワーキンググループ、および DBPL サブワーキンググループでの議論が有意義でした。委員・オブザーバの方に謝意を表します。また *Quixote* の設計開発の過程での議論は、DOOD の技術的問題点の検討を深めました。同プロジェクトのメンバーの方に感謝します。最後に、講演のわかりにくい内容を、辛抱強くわかりやすくまとめていただいた須貝昌子さんに深謝します。

参考文献

[1] Kim, W., Nicolas, J.-M. and Nishio, S. (ed.): *Deductive and Object-Oriented Databases*, North-Holland, 1990.

[2] K. Yoshida, "The Design Principle of the Human Chromosome 21 Mapping Knowledgebase (Version CSH91)", Internal Technical Report of Lawrence Berkley Laboratory, May, 1991.

[3] K. Yokota and S. Nishio, "Towards Integration of Deductive Databases and Object-Oriented Databases: A Limited Survey", アドバンスト・データベース・システム・シンポジウム, 京都, 12/7-8, 1989.

[4] 横田, 西尾, "演繹・オブジェクト指向データベース", 情報処理, vol.31, no.2, 1990.

[5] 横田, 森田, 宮崎, "オブジェクト指向データベース・プログラミング言語", 『情報処理』, vol.32, no.5, 1991.