Vol. 33 No. 10　　情　報　処　理　　Oct. 1992

報　告

パネル討論会

永続的プログラミング言語とオブジェクト指向データベース†

第85回 データベースシステム研究会
第4回 プログラミング—言語・基礎・実践—研究会　合同報告

パネリスト
大堀　淳[1]，　小野寺民也[2]，　鶴岡　邦敏[3]
布川　博士[4]，　牧之内顕文[5]
司会　安村　通晃[6]

データベースシステムの研究開発においては，従来からの階層データベース，関係データベースやネットワークデータベースに加えて，いわゆるオブジェクト指向データベースが盛んになってきている．オブジェクト指向データベースが盛んになってきた理由としては，一つはテキストのみのデータベースからマルチメディアデータベースへと取り扱う対象の広がりがあるが，もう一つは，データモデルとしての自然さがある．

オブジェクト指向データベースの研究も，プログラミング言語におけるオブジェクト指向の研究からの影響を受けて活性化したことは否定できない．

データベースシステムの言語としての*インターフェイスは，永い間，データベースシステム自身が専用の検索言語をもつか，あるいは，汎用のプログラミング言語からの呼び出し口をもたせるか，のどちらかであることが多かった．やや複雑な処理をしたい場合には，プログラミング言語のほうで記述する必要があった．オブジェクト指向データベースの研究開発の展開と合わせて，データベースシステムの中にも一般的なプログラミング言語へと統合の要求が出てきた．

一方のプログラミング言語といえば，近年では，汎用の手続き型言語よりも，論理型言語，関数型言語，オブジェクト指向言語などの，宣言型のプログラミング言語に研究開発の中心がシフトしている．その中で，一つは関数型言語の研究を通じて，データ型のもつ重要性が明らかになってきた．もう一つは，プログラムの記述性やモデル化の容易性からオブジェクト指向言語の研究が活性化している．

オブジェクト指向言語の場合，(1)データ構造とその操作の一体化によるデータアクセスの隠蔽化 (Encapsulation)，(2)クラスの定義とその継承 (Inheritance) による抽象化と再利用，および，(3)メッセージパッシングによるオブジェクトの起動，などの特徴を一般的にはもつ．

オブジェクト指向言語では，特に，データをオブジェクトとして自然にプログラム中に含むことができるため，実行中に作られたデータを後で再利用するために残しておきたい，という要求が出てくる．これが，永続性 (Persistency) の要求である．

従来言語でも，ファイルという形では永続的なデータをもつことができた．しかし，ここでいう永続性の要求とは，特定のデータ型だけでなく，すべてのプログラム中の値とその型が永続的に扱えることである．このような永続性をもったプログラミング言語を永続的プログラミング言語と呼ぶ．

この言語は，データベースシステム（特にオブジェクト指向をベースにしたもの）に汎用の言語機能を取り込んだものと限りなく近い関係にある．つまり，遠景からみれば，両者の類似性は明らかである．しかし，近景から，詳細に両者を眺めれば，プログラミング言語をベースにしたものと，データベースシステムをベースにしたものとでは，いろいろな点で違いもはっきりしてくる．

☆編集委員会ではインタフェースと表現していますが，著者の希望によりインターフェイスとします．

† 日時　平成3年10月4日 (金) 15：15～17：00
場所　機械振興会館6階66号室

1) 沖電気，2) 日本アイ・ビー・エム，3) 日本電気，4) 東北大学
5) 九州大学，6) 慶大

以上述べた観点から，パネル討論の中ではデータベースシステムの立場と，プログラミング言語の立場から，双方の要求事項をとりあげ，その共通点と相違点を明らかにしていくことにより，「永続的プログラミング言語とオブジェクト指向データベース」の関係を明らかにしていく．

具体的には，次のような点を議論していく：

（1） オブジェクト指向データベースとは
（2） オブジェクト指向データベースの実例
（3） 主記憶データベース*
（4） 型理論からみたオブジェクト指向
（5） プログラミング言語からみたデータベース

安村通晃

参考文献

1) 特集「オブジェクト指向データベースシステム」，情報処理，Vol. 32, No. 5 (May 1991). (たとえば，増永良文，次世代データベースシステムとしてのオブジェクト指向データベース，など).
2) Atkinson, M. P. and Bunenman, P. P.: Types and Persistence in Database Programming Languages, ACM Computing Surveys, Vol. 19, No. 2 (1987).

はじめに

（司会）・安村　本日は，データベースシステム研究会とプログラミング研究会との共催で，「永続的プログラミング言語とオブジェクト指向データベース」と題して，パネルを行います．パネラーは，順に九大の牧之内先生，日電の鶴岡さん，沖電気の大堀さん，日本 IBM の小野寺さん，それに，東北大学の布川先生です．私は，司会を務めます慶應大学の安村です．では，さっそく牧之内先生，お願いします．

牧之内　永続データに関する研究が何かなされているかと思い，情報処理学会第43回の発表内容をざっと見てみました．そうすると1件しかありませんでした．「並行オブジェクト指向言語における永続オブジェクトの実現」．126件中の1件ということでなんとなく寂しい．

* 大容量の主記憶をディスクの代わりに見立てることによる，データベースの新しい構築法の一つである．

なぜ永続的プログラミング言語か？

なぜ永続的プログラミング言語が実用上重要か．第一にファイルを扱わないですむ実用的プログラムの存在を私は知らない．だから実用的なプログラムで，永続データというものを扱わないものはないだろうというのが私の信念です．

RDB というのはレコード処理，事務処理分野を応用分野として考え出されました．CAD など複雑なデータ構造を扱うには適当でないということで OODB など次世代 DB 論議が盛んです．それに対してプログラミング言語のほうはレコードファイルの read/write しかないが事務処理分野以外のことにもいろいろ適用されています．CAD のような応用でファイルを使い複雑なデータ構造を処理しなければならないとき，現行のプログラミング言語を使ったのではプログラマは苦労するのではないでしょうか．

現行のプログラミング言語にはファイルデータを扱ううえでの難点があります．つまり高水準のプログラムデータ構造とファイル構造との間に存在するギャップが非常に大きいというわけです．たとえば「データとアルゴリズム」なる本を見ると必ず出てくるのは二分木ですね．二分木による辞書構造，たとえば辞書を作ってみよというと，だいたい皆さんの頭にはアルゴリズムがすぐ浮かんでくるはずです．それじゃそのファイルによってその二分木を作ってみよというと，なかなか作れないのではないか．

二分木による辞書構造を普通のプログラムデータで作りますと，ポインタによる木構造で作ります．それをファイル構造で作ってみよと言われても普通の平均的なプログラマだとまず無理だと思います．これが僕のいうギャップなわけです．このような複雑な構造を永続化するということが非常に重要だと思います．

永続的プログラミング言語というのは何か．「永続データの操作範囲が揮発データ—揮発(volatile)というのは普通のプログラムデータですが—の操作範囲とまったく同じであるようなプログラミング言語」．つまり揮発データに適用できる操作がほとんど同じように永続データにも適用できるということができるような言語が永続的プログラミング言語だと思っています．

それに似たような概念がデータベース分野にもあります．データベースプログラミング言語(DBPL)．これはデータベース操作が可能なプログラミング言語のことです．リレーショナルデータベースであれば SQL を埋め込むことが可能な COBOL, Fortran, C などは DBPL であろうと考えてよい．そうするとファイルを扱うことができるのはファイルプログラミング言語ということになります．ALGOL 60 は，もともとファイルを扱う規定がなかった．一番最初に述べたように，これでは実用的なプログラムが書けなかった．したがって死語になってしまった．

この三者を比較すると，永続的プログラミング言語では，データ型というのはほとんどすべてが永続的にできます．が，DBPL，たとえば RDBPL であればリレーション，タップルしか永続化できない．しかも操作は SQL でできることしかない．一方ファイルプログラミング言語ではファイルのレコードの永続化のみである．

しかし永続化されたデータは大量になります．何か残しておけるとなると，みんな要らないものでも残すというのが人間の性質ですから，それから残したものは共有される．後日あとで再利用されるということが起こる．

すると，昔作られた永続データ群のうちで，いま自分に必要なデータを探してそれを利用する場合が出てくるわけです．ここで問合せ言語というのがどうしても重要になってくる．たとえばそのデータはいったいどんな構造をしていたのかということを思い出さなければいけない．また，そのデータを勝手に変更していいのかどうかというのも共有という問題に係わってきます．そういう意味でデータベース管理システムが必要とする機能すべてが，一つの言語の機能としてうまく統合されないといけない．そうなりますと言語の，たとえば最適化ということも非常に広い意味になってくるわけです．いままでの言語の最適化だけではうまくいかない．

DB 分野における次世代論がいま非常に盛んですけれど，プログラミング言語における次世代論はあまり聞かれないのは少しさびしい気がします．永続的プログラミング言語を次世代プログラミング言語として位置づけ，もっと活発な研究を期待したい．

司会 では，続きまして日電の鶴岡さん．

鶴岡 私のところではオブジェクト指向データベースの Odin というのをやっておりますので，その開発経験を述べます．

オブジェクト指向データベースの機能

最初は，揮発データ，永続データに関してです．RDB だと，永続データというのはリレーションという特定な形で蓄えており，ある代数系を作っています．これは，たとえばグラフィクスのデータ構造とそれを扱うサブルーチンとの関係に似ていて，一見，自然に思えます．しかしよく考えてみると，永続性というのは特別な性質ではなく，あらゆるデータがもち得る性質です．ですから揮発データに関するプログラミング方法論と永続データに関するそれが違っているのはおかしい．永続性の有無によって応用プログラムの記述を同一にするには，言語側のモデルとしてデータベース機能をもたないとだめです．要するに，言語側のモデルを拡張してくださいということです．

ただ，データベースというのは共有という性質があるので，主記憶側でもっている複雑な構造を永続側に持ち込む場合には，利用者ごとの見方，ビューの提供が必要でしょう．

そこで，プログラミング言語側への要請ですが，第1に型とクラスです．型は型として存在する一方で，実行時に動くオブジェクトとしてクラスも必要ではないでしょうか．DB では，もともとクラスに対応するものとしてスキーマという概念があります．DBMS はスキーマを使って動いています．なぜ必要かというと，たとえばクラスが保有する定義情報を実行時に参照して動くツールとして，ブラウザ，インタープリタなどがあります．また，データベースというのはデータの寿命が長いことを前提にしますので，その上にたくさんの応用プログラムを作ります．その場合，クラスの構造が変わったときにプログラムを全部再コンパイルするとか，データを全部再構成するのは非常に大変です．それで，性能に影響がないかぎりは，実行時にクラスを参照するようにしたいわけです．

2番目に，型構成子や実行時に生成される型です．DB の場合，たとえば社員の集合があって，その中で条件を満たすものを選択（select）することがあります．この場合，選択したものを別の型として扱いたい場合があります．そのほかに，二つの型を結合（join）した型があり，これは実行時に動的に作ります．このように，select とか join とかを定義すると，型の名前のかわりに型構成子を指定する機能が必要になります．クラスを定義するときにも，その型を型構成子で与える機能が必要です．たとえば，社員集合などです．

3番目，ビューという概念が必要だと思います．共有データがあり大勢の人が使いますから，個々の人が個別の見方をしたい．そのためにビュークラスというのが必要になります．このインスタンスは，他のインスタンスを見ているようなものです．たとえば，結合インスタンスは，社員と会社を合わせたオブジェクトとして見えます．社員と会社の両方のインスタンス変数が一緒に見えるもので，その中で可視変数とか可視メソッドを使って制限をすることができます．さらに付加メソッドもあります．

さらにトランザクションですが，これはデータベースに本質的なもので，やはり言語側の構文として提供できないといけない．単にトランザクションという関数だけを書いておきますと，プログラミングの仕方によって，たとえばトランザクションをコミットもロールバックも実行しないで抜けてしまうことができます．Odin の場合は，トランザクション文という形でブロック構造になっていて，ここから出るときは拡張ブレーク命令を使用します．これはロールバック命令に相当し，コミットのほうは，トランザクション文の終わりにシステム側で発行します．

5番目に，集合的な検索機能についてですが，揮発オブジェクトか永続オブジェクトかによって構文が変わらないことが前提です．ですから言語の中で，インスタンス集合の中で条件を満たすものを探す，などの構文が揮発オブジェクトに対しても書けないといけない．Odin では，C++ の構文を SQL ふうに拡張しています．

実現上の問題で，いまの言語処理系に対する不満があります．クラスの定義情報をソース形式で管理し，ヘッダファイルをインクルードして使うのは非常に大変です．これは DB として管理しておくと良いと思います．

メモリ管理に関しても，何でもヒープに作ってこれがどんどん大きくなってしまい，そのうちメモリがスワップ領域をオーバフローして落ちてしまうことが起こります．こういうやり方はまともな DB の使い方としては許されません．言語処理系と DBMS のメモリ管理をうまく統合して，安定した動作を示すことが必要です．

エディタ，コンパイラ，デバッガ，といった開発ツールがファイルベースで作られているという問題もあります．本来，これらはファイルを意識しない仮想的なツールになっているべきではないか．つまりエディタでしたら，ソースメソッドクラスというものがあり，このクラスのインスタンスを編集するようなオブジェクトエディタが存在する．これはファイルからの読み込みとか書き戻しを意識しないで，仮想的なオブジェクトだけを扱っていればいい．コンパイラもそうです．ソースメソッドクラスのインスタンスを実行形式メソッドクラスのインスタンスに変換するオブジェクトトランスレータと考える．

最後に言いたいのは，OODB と PPL*を統合して一貫したプログラム開発環境を提供する必要があるということです．

司会　次に入る前に短い質問を1，2問受けたいと思います．

箕原(慶大)　クラスの動的参照に関してですが，動的なアプリケーションによって使われるクラスでは，そのクラスの中の特定の名前とかも初めて実行時に分かるわけですね．そういうような環境において鶴岡さんのシステムではどのように解決なさっていますか．

鶴岡　Odin ではクラスはオブジェクトであって，クラスがインスタンス変数の名前などの情報をもっています．対象とするクラスが実行時に決まるような処理を記述したければ，クラスを参照しながら動くプログラムを書くという前提です．

司会　鶴岡さんのシステムは C++ ベースですが，クラス変更のとき，再コンパイルが必要ですか？　それともインタープリタで動いているので，その辺は大丈夫なのでしょうか？

鶴岡　クラス変更命令というのが別にあって，

* PPL: Persistent Programming Language

それはインタープリティブに実行できます.

鈴木(電気通信大学)　クラスはそもそも永続的だと思いますが,なぜ永続クラスというふうに書く記述があるのでしょうか?

鶴岡　プログラムが動いている間だけ定義されるクラスとして,揮発クラスがあるためです.

鈴木(電気通信大学)　でも,プログラム中に含めてしまうと,名前が衝突したときどうなるかとかが問題になりますね.そうするとコンパイルが環境に依存してコンパイルできたりできなかったりする.いろいろ困ったことが起こりうる.

鶴岡　揮発クラスの場合は,プログラム自身がそのクラスを作り出します.そのクラスはプログラムの中だけで見えます.

司会　質問はこのくらいにして,それでは,IBM の小野寺さんにプログラミング言語の立場から.

小野寺　今日言いたいことの一つめは,OS やコンパイラといった普通のシステムプログラムの分野でも,データベース的な考え方,特に「トランザクションによるプログラミング」が必要だということです.いままでは,こうした分野のトランザクションは,はっきりとそれとは自覚されないまま,あるいは,アドホックな形で,実現されてきましたが,データベースの分野における本格的な研究成果を採り入れつつ,きれいにプログラミングする必要があると考えています.

データベースの考え方

たとえば,UNIX の /etc/passwd はデータベースであるといえます.このデータベースが,version 7 のころの UNIX でどのように実現されていたかといいますと,普通のテキストファイルなわけです.「問合せ」するときは,そのテキストファイルをリードしてパーズします.「更新」するときは,エディタとかストリームエディタといったテキスト処理用のツールを使います.ナイーブなデータベースですから,更新があるたびにテキストファイル全体が書き替えられます.

このデータベースで,トランザクションの第1の性質である「シリアライザビリティ(直列可能性)」がどう実現されていたかといいますと,更新するときに排他ロックを取ります.creat( )というシステムで代用していたはずです.次に,トランザクションの第2の性質である「リカバビリティ(障害回復性)」はどう実現されていたかといいますと,(1)まずオリジナルのコピーを作って,(2)更新はコピーのほうに行って,(3)完全に更新が終わったところで,原子的であるところの mv 命令によってオリジナルを置き替える—という形を取っています.こうして曲がりなりにも「データベース」を実現していたわけです.今日,プログラミングの対象はどんどん拡大し,トランザクションが数多く潜んでいる領域に入り込んでいるので,これをもっときれいにプログラミングしたいと思っているわけです.

今日言いたいことの二つめは,こうしたデータベースは小さいということです.たとえば,私が日ごろ使用しているマシンの /etc/passwd は 1K くらいですし,研究所にあるほとんどの UNIX マシンのアドレスを保持している /etc/hosts は 15K ぐらいです.あと,junet とか usenet を読むとできる.newsrc というファイルの大きさは,私の場合,22K ぐらいです.もう一つ,プログラミング環境のほうから例をあげます.ソースコードブラウザなどのツールから参照されるプログラムデータベースの大きさですが,私の使用しているシステムでは,InterViews という3万行ぐらいの GUI ライブラリを格納した場合,だいたい 40M ぐらいになります.では,本格的なデータベースの大きさはいったいどのくらいなのかといいますと,たとえば,TWA という航空会社の切符予約システムの大きさは 2G です.トランザクションは1日700万回で,バーストレートでは1秒間に100回ぐらいです.ちなみに,これは1983年ごろの話です.いまでは,もっと巨大化していると思われます.

メモリベースのデータベース

今日言いたいことの三つめは,小さなデータベースの実現にはそれにふさわしい手法があるのではないかということです.「オブジェクト指向データベース宣言」の全項目を満たす必要はないはずです.そこで,メインメモリデータベース(MMDB)という手法があるのですが,これが有

望なのではないかと思っています．ひとことで言うと，データベース中のデータをすべて仮想記憶上にもつわけです．ある意味では one level storage に近いものがあるのですが，最大のメリットは，問合せが非常にチープになることです．ここで突如としてプログラミング言語と関連した話になるのですが，データ（オブジェクト）のネットワークが仮想記憶上にでき上がるわけですから，これをプログラミング言語におけるデータ構造としてそのまま取り扱えればプログラマにとってハッピーであるわけです．問題はトランザクションをどう実現するかということですが，たとえば，Birrell という人は，チェックポイントとリドゥログ（redo log）を用いる方法を提案しています．

先の議論で，MMDB とプログラミング言語が結び付きました．さらに OS の仮想記憶管理システムと融合させることを考えます．つまり，データベースの2次記憶管理を OS の仮想記憶管理に任せようというわけです．少なくとも，小さなデータベースシステムではこれで十分です．洗練されたクラスタリングは不要でしょう．あるいは，せいぜい，仮想記憶システムにおけるローカリティの問題という視点でとらえるので良いと思います．すると，ここで，長い研究の歴史を誇るガーベジコレクションの成果などが活きてくると思います．

結論を言いますと，(1)MMDB と (2)プログラミング言語と (3)OS の仮想記憶管理の三つをうまく融合させることにより，小さなデータベースを効率良く実現し，トランザクションをきれいにプログラミングしましょうということになります．幸い，仮想記憶のインターフェイスを外に出した OS も最近ポピュラーになりつつありますので，こうした研究方向は現実的なものであると確信しています．

司会　では，前半部のディスカッションに入りたいと思います．

大堀　牧之内先生の発表に非常に共感するものがありましたので一言．先生のポジションは，「永続言語の研究は実用上重要である」ということですが，同感です．ところでこれを定理としますと，「永続言語の研究は理論的にも重要である」との系が導けます．証明は簡単でして，理論的研究の一つの重要な役割は，永続言語のような複雑なシステムをちゃんと設計可能にするということだからです．そのような話を今日はしようと思ってまいりました．

ただ，用語の定義については多少意見を異にしております．永続言語の定義は同じなんですが，DBPL は，

DBPL＝PPL＋DB用に拡張された強力な型システム

ととらえたいと思います．そのような型システムはまだありませんから，望ましい DBPL は，今後の研究成果の系列の上界として実現されると考えております．

箕原(慶大)　永続性を考えるうえで，たとえばエディタとかデバッガとか，そういうツール自身もなんらかの他のプログラムによって操作される対象となるわけですね．そうすると，実は永続システムというのは非常に総合的なシステムだと思います．この総合的なシステムにおいて，型というのはたとえばエディタのセマンティックスにデバッガのセマンティックスというのをどこら辺まで扱えると考えますか．

大堀　現在はうまくいってないと思います．たとえばもっと単純な例で，UNIX のコマンド言語がありますが，あれは型がないですね．どうして型がないか，一番問題なのは，お互いに他のシステムの型を理解できないという点だと思います．

データモデルとビュー

北川(筑波大)　データベースシステムという概念がファイルシステムから出てくるにあたって，データ独立性というアプリケーションとデータを基本的には切り離そうということが重要な概念として認識されてきました．その結果としてリレーショナルモデル，単純なデータコードでデータをもっていて，アプリケーションはそれを適当に見方を変えて使えるようになるというような見方もあります．あるいはデータベース管理システムを作るという立場からいうと，簡単な構造に限っておいて，大量データがきたときに最適化なりインデックスみたいなものを使えば，content addressable なメカニズムを作りやすいという背景があ

るかと思います.

ですから共有されるデータを含んだデータベースというものと，アプリケーションスペシフィックなデータがとにかく永続的に存在すればいいというところにギャップがあるのではないか．牧之内先生から，基本的には非永続的のものと永続的のものをユニフォームにみえるところが一種の理想なんだというお話がありました．しかし，DBMS（データ管理システム）という立場からいったときに，従来からのデータと応用を切り離そうという考え方だとか，そういうものへの考慮というのは，どのようにお考えでしょうか.

牧之内　いまご指摘のことは先ほどのモデル論に関係する重要なことです．普通のプログラミング言語で自分が，たとえば二分木を作る．自分で蓄えておいて，あとからそれを使おうとすると使えない．ましてや他人の作ったものは再利用するのは難しい．データベースではモデル論があり，スキーマでちゃんも定義しておきます.

永続データの共有を考えると，永続的プログラミング言語でも，もっとモデル論が入って良いという気がします.

モデル論がないと，たとえばCの揮発データがみんな永続的にできることとなり，混乱が起きる.

そんな理由から永続プログラミング言語にもやはりなんらかの新しいコンセプトが入らないといけない．それはやはりモデル論ではないかという気がします.

北川(筑波大)　コンテキストによってどの程度アプリケーション型のデータの見方を，どの程度のバラエティをもたせたいかということが，そういう環境によって大分違ってくるだろうと思います.

牧之内　データの見方，DBMS でいう「ビュー」について一言付け加えます.

永続データが存在する．その永続データを検索し，操作するのが応用プログラムです．そのプログラムの操作そのものが「ビュー」ということになるのではないでしょうか.

だから，プログラムで自分なりのことをやりたいわけですから，どうしても見方は偏るわけです．アプリケーションに依存するわけです.

横山(ソニー)　永続データと揮発データを同じように扱うために，テラバイトのオーダでアドレッシングができるような CPU をもってきて，ディスクのスペースも全部そのアドレス空間に割り当てるというのはどうでしょうか？　細かいことは抜きにして，永続的なディスクの中に入っているものも同じように扱えるのではないかと思いますが，その方向で何か考えられないでしょうか.

小野寺　それは，基本的に先ほど紹介したメインメモリデータベースといっていいと思います．もともとデータベースの専門家は，小さな仮想記憶で大きいデータをどう扱うかというのに腐心してきたわけだから，仮想空間が大きいとなれば，問題のいくつかの部分はすでに解決されています．ただ，巨大なデータベースを作ろうという場合は，クラスタリングなどの技法を無視できないと思います.

仮想空間：一つか複数か

鶴岡　one level store みたいなものができたときに，仮想空間というのは一個でいいんでしょうか．たとえばネットワーク全体の仮想空間を作ったときに，それは最初に定義された1個の仮想空間にマッピングできるか．それに入っていないものが出てきたときにどうするかです.

大堀　それは本質的な問題だと思います．種々の異なった環境があって，それらがデータをシェアするとなると，one level store だけではうまくいかなくて，外部データを取り込むメカニズムが必要になると思います．型理論からみた永続データの本質も実は，それらデータが，あらかじめ設定された型の環境の外部にあるという点です.

牧之内　いま小野寺さんがいわれた one level store という話がありますね．ああいう大きなアドレス空間の中にファイルをうまく組み込む技術は，僕の意見では永続的プログラミング実現のためのインプリメントの一つの技術だと思います.

ただ，組み込み方がいろいろあって，たとえば Mach の external pager の考え方では，ファイルは外部に存在するわけです．それをマップするわけです．マップするわけだから，大堀さんが言われたように，外部にあって，必要なときにマップして，そして計算可能空間にデータをもってきます．それを使えばデータの永続化は非常に簡単にできるわけです.

実は私のところでもそれをインプリメントしています．ただ，データのリカバリやコンカレントコントロールをどうするかというのが，難しい．しかしそれが真の永続的プログラミング言語のインプリメンテーションの第一歩だという気はします．揮発データと永続データの扱いが同じであるというのは効率的にもそうでなくてはならない．そうするとやはり今いったようなことをうまくハードウェア的にもサポートするような仮想空間をうまく使うことが重要だという気はしています．

箕原(慶大)　いままで永続的プログラミング言語に欠けていた議論として，1人で使っていたものを他の人に対してどう使えるかというような，使用という概念が永続的プログラミング言語ではなかなかはっきりしてこなかったということがある．それがデータベースを中心に考えてきた者にとっては，データベースには基本的なカタログとかシステムテープがあって，データを共有するときどんなものがあるのか，どういう状態であるのかというのは非常に分かりやすくユーザに提供されています．多分私は総合的なシステムであるかどうかというのは，そのような使用というものが提供できているか，あるいは単なる言語だけで終わっているか，そういうものだと考えます．

鈴木(電気通信大学)　仮想空間が1個だけでいいかという話ですが，たくさんなければいけないというのが僕の意見です．実用的なものを考えるのでしたら，複数の記憶空間を考えなければいけないし，複数の記憶空間を考えるのでしたら，型と永続が直交しない場面が出てくるんだと思います．

牧之内　複数の仮想空間をもつ必要があるというのはよく分からないのですが，要するにファイルをマップできればいいわけでしょう．

大きな仮想空間があって，ある部分，たとえば1メガから5メガまではこのファイルをマップします．そしてそういうことが意識できて，アドレスが1メガから5メガの間を操作することはマップされたファイルをいじっているというふうにできればいいのではないですか．

鈴木(電気通信大学)　オブジェクトを作るときに，このファイルの上にオブジェクトを作るとかということをプログラミング言語で指定できるようになっていないといけない．もしくは自動的にするのでしたら，それをするためのメカニズムはプログラミング言語に道具として用意して利用者に開放しておかないと，自動的にどこか魔法の場所を作られるというのではうまくいかないと思います．たとえば，PS-Algol だと，データベースに作られたオブジェクトから read されているオブジェクトでは自動的にデータベースに入るようになっていますが，あれが2個データベースを開いて，共通したオブジェクトがあったとき，そのオブジェクトはどっち側のデータベースに入ればいいかという問題があって，Buneman 先生に聞いたら，パーシステントルートがあると聞いてびっくりしました．そういうものがあるのでは，データも持ち運びするのに困ると思いました．ですから複数の記憶領域を開放して，しかもプログラミング言語の中でマニュアルに利用者が指定できるようにしておかないといけないと思います．

本田(慶大)　ファイルネーム，あるいは名前空間とプログラミング言語をどういうふうにお考えになっているか，大堀さんにお聞きしたい．いまのところあまりマッチしていないとおっしゃったと思うんですが．

大堀　プログラミング言語の理論的な側面では名前の取り扱いは，トリビアルですね．すべて λ バインテイングで名前のバインテイングをする．ただ，もちろんそれだけではすまないはずで，外部のデータ，たとえばネットワーク上のデータの操作には別なメカニズムが必要だと思いますが，それについてはあまり研究されていません．

なぜオブジェクト指向か？

松岡(東大)　牧之内先生の研究のモチベーションは正しいと私は思っています．なぜかというと，たとえばかつてはユーザインターフェイスも「シーハイムモデル」というものがあって，それはまさに DBMS を範としてユーザインターフェイスマネイジメントシステムを作ろうとしたんですね．それによって，なるべくアプリケーションとユーザインターフェイスの間のデータ独立性を保とうとしたわけです．ところがそれでは少なくとも当時の技術レベルではうまくいかなくて，結局たとえば Interviews ですとか，NextStep とか，MacApp のような，いま牧之内先生がおっしゃっ

たような方法でオブジェクト指向の UI ツールキットを構成しておいて，ユーザが新しいアプリケーションを組みたいときはそこにインヘリタンスを使ってカスタマイゼーションを行うといったことが主流となったわけです．それによってポータビリティを維持しながら，応答性が良く，かつそれぞれのアプリケーションに容易にカスタマイズできる，ということが達成できたわけです．

そこで質問ですが，永続的プログラミング言語でもそのようなアプリケーションに対してカスタマイズが可能であるというのは，オブジェクト指向であることが本質だとお考えですか．つまり，UI では，C++ とか Smalltalk とかのオブジェクト指向言語によってクラスライブラリという概念が出てきて初めて可能になった．果たして永続的プログラミング言語でもオブジェクト指向言語でクラスライブラリが構築できるということが本質的だとお考えでしょうか．

牧之内 とりあえず世の中にあるコンセプトを眺めてみて，オブジェクト指向モデルというか，パラダイムというのは多分役に立ちそうだという気はしています．二つの理由で役に立つ．一つの点はなんとなく役に立ちそうだということ．もう一つは，これはオブジェクト指向パラダイムが永続的プログラミング言語に本質的に重要になると私が思う概念ですが，データと手続きの encapsulation が可能ということです．

先ほど申しましたように永続データがいろんなところで勝手に作られるわけです．あとでデータを使おうとするときは，そのデータにどんな機能があるかと，どんな動きをするかということが分かれば，それで十分なんです．つまり，たとえばテレビジョンを見るのにボタン操作の機能が分かればいいのと同じことです．そういう意味でオブジェクト指向の encapsulation という概念は僕は非常に役に立つ概念だという気がしているんです．

それ以外，オブジェクト指向パラダイムで提案されている構造的側面は，永続的になるとなんか害を及ぼすのではないかという気がするのです．ある構造に合っているとあるとき認識するのは，必ずしも明日は正しいとは限らない．永続データは 1 年前に作られたかもしれない．

田中(松下電器) 先ほどの，大きな仮想記憶領域をもっていれば良いという話をちょっと蒸し返させていただきますけれども，ネットワークワイドで，永続言語を使って，共有するような状況が今後考えられてくると思いますが，その場合には僕はメモリマップを行うだけではうまくいかないと思っております．なぜかというと，アーキテクチャの違うマシンが必ず入ってくるわけです．

もう一つは実際問題としてパフォーマンスを考えたときに，マシン間で仮想記憶をシェアしてしまったならば，ネットワークを介した仕事がガンガン起きて，現実的なシステムとしてはきつい．CPU は速くなるけれどもネットワークは遅いわけですから，ということで実際にはネットワークワイドな仮想記憶空間をもってシェアしようというのは，ここ何年か，というか，10 年以上はちょっとうまくいかないような気がします．

それよりは，それぞれの永続オブジェクトに対してネーミングを行って，そのネーミングを管理するサーバを介してオブジェクトという形でシェアするというインプリメントが正しいのではないか．これからそういう方向で考えないとインプリメントは辛いのではないかと思います．

小野寺 オブジェクトを，仮想記憶機構を通じて共有しても，サーバを介して共有しても，基本的な問題は一緒だと思います．まず最初の「ヘテロな環境ではうまくいかないのでは」という質問ですが，仮想記憶に張りつけるときに，単にページをそのまま持ってくるのではなく，XDR に相当するようなフィルタをかければ良いのではないでしょうか．

次に，パフォーマンスの問題ですが，オブジェクトのネットワークがマシンの境界を越えてでき上がり，マシン間でオブジェクト転送がガンガン起こり得るのは，いずれの方式でも同じです．ここで，重要になるのは，プロファイラだと思います．いいプロファイラを使って，オブジェクトの配置をある程度最適化する必要があるでしょう．それとは別に，ネットワークが高速になることも十分期待できると思います．

司会 ここで後半戦に入りたいと思います．では，まず大堀さんからお願いします．

型理論からみたデータモデル

大堀 タイトルは「型理論に基づく次世代データモデル実現の可能性」．サブタイトルとして，

先日の仙台での Dana Soott の話*のタイトルをもじって「データモデルをタイプシステムで置き替えられるか」と考えたのですが，もうちょっとリファインして「データモデルとタイプシステムは統一できるか」としたいと思います．

前提は，単なるタプルだけではなく，いろいろなデータを扱うようなデータベースがあってほしい．しかもそれが，シェアされた永続データとしてプログラミング言語から自由に使えるようになってほしいということです．

そのようなデータベースに対して，関係データベースで経験したように，種々の実現技術が蓄積され，それが次世代 DB として発展していくためには，やはりそれがデータモデルとして確立しなくてはいけない．そのためには少なくとも，データ操作言語の形式的な定義とその厳密な意味論が必要です．そのようなモデルを実現するステップは，(1)次世代 DB の望ましい機能の選定，(2)数学的手法を用いた機能間の関係の分析，(3)十分に一般的なパラダイムのもとでのそれら機能の実現，となっていて欲しいと考えます．現在，次世代のデータベースの望ましい機能はいろいろ議論されておりまして，この(1)の段階としては非常に健全な状態にあると思います．その次の段階としてどうしても期待したいのは，厳密なフレームワークでの機能間の関係の分析と，十分に柔軟性のあるパラダイムに基づいたそれら機能の実現です．

可能なアプローチは種々あります．たとえば構造化された項を含む述語論理を用いるのも一つの有力なアプローチだと思います．それら種々ある可能性の一つとして，私は，λ計算の型理論とデータモデルの統合という方法に注目しています．λ計算の型理論は，汎用の計算能力とデータ構造の制御機構です．この枠組に，データの永続性，データモデル，メソッド，インヘリタンスなどをうまく採り入れるのが課題です．

まず，永続データについて，先に言いましたが，型理論的にみた永続データの本質は，プログラムの静的環境の外部にある，ということです．永続データの研究の中で得られた重要な概念に直交永続性があります．外部の永続データを扱うメカニズムは，その他の言語機能と独立性が高いという洞察に基づいています．永続言語研究の貢献は，そのようなメカニズムを確立したことです．

永続性に関しては，課題はいろいろありますが，ある程度見通しがついていると思います．もっと大きな問題として，型システムをデータモデルへ拡張するという研究を今後しなくてはいけないのではないか，というのが私の研究の主な動機です．やらなければならない研究はいろいろありますが，(1)複合オブジェクト操作システム，(2)データベースの設計理論，(3)大容量型 (bulk data type) についての理論，(4)オブジェクト指向プログラミングの機能の導入，(5)実装理論，の5つをポジションペーパーにあげておきました．

まず複合オブジェクト操作システムですが，関係代数に相当する厳密な定義が必要であると思います．これがあれば複合オブジェクトデータベースに対して，エレガントで簡潔な問い合わせ言語が作れて，その最適化理論が生まれる可能性があると思います．さらにビューも，オブジェクト同一性をサポートするような操作系ができれば，実現できると思います．あとはデータベースの設計理論と，大量のデータの操作に関する理論，これは明らかに重要ですが，これらも実はこの(1)を前提にして初めて可能になるわけです．

以上の要素に加えて，オブジェクト指向プログラミング機能，実装理論などを含んだ整合性のある体系を，型理論を一つの道具として使って作れると思っています．

結論ですが，型理論の研究で開発されたいろいろな技術を利用することによって，単なる情報検索システムにとどまらない次世代データモデルの構築が可能になると信じております．Dana Scott のサブタイトルの引用を続けまして，さらにリファインされたサブタイトルとして，「データモデルとタイプシステムは，いつユニファイされるか？」としたいと思います．

本田(慶大)　それはいつですか．

大堀　引用の続きだと2000年ということになっていますが…

司会　では，引続き布川先生には言語パラダイムの点から．

☆ "Will Logicians be Replaced by Machine?" by Dana S. Scott, TACS '91.

布川　タイトルをつけようと思ったんですが，なかなか思いつきませんでしたので，ここにありますように，計算モデルというかプログラミング言語という立場から，データベースや，さまざまなソフトウェアのシステムをみてみたいということです．私はもともとプログラミング言語，特に関数型言語をやっておりました．

計算パラダイムとデータベース

最近の中心的な問題意識というのは，人間にとって環境としての分散システム（これをわれわれのグループでは分散環境と呼んでいます）をどのようにして構築するかということです．

具体的には，計算モデルの立場から，非常にフレキシブルな計算モデル，協調型計算モデルをやってます．またユーザインターフェイスの立場からはメタファを提供する分散ユーザインターフェイス，ネットワークの立場からは，オブジェクト指向データベースを用いたネットワーク環境マネージメントをやってます．

当然，私1人では力が及びませんので，多くの先生方に教えていただきながら研究をやっております．そのうちでもデータベースに関連してお教えいただいている，図書館情報大の増永先生との議論ではさまざまな意味で，カルチャーショックを受けました．

データベースとプログラミング言語の領域では同じようなことをやっているようで違うところもある，といったことでしょうか．そんなカルチャーショックを受けた一人のプログラミング言語の立場の人間として，データベースがどうみえたか，また，今日のキーワードである，永続化というものがどうみえたかをお話させていただきます．

まずはじめに，計算モデルということですけれども，過去さまざまな計算モデル的な分類の中には，構造化というのがあったし，抽象データというのがあって，オブジェクト指向というのがあるということになります．また，手続型，非手続型というのがあったし，いまはやっている分散協調型とかがあります．型については，あるものないもの，階層があるとか，高階，型変数の有無とかあるわけけです．関数型，論理型という意味論からの分類もあります．これらを言語の高級性と呼んでみます．

でも実際は，こんな話だけでは使えなくて，ライブラリの多さとか関数の多さとか，特定の分野の応用のために，その分野用の機能が入っているかということがきわめて大事にされるときがあると思います．これを言語の高機能性と呼ぶことにします．

つまり，プログラミング言語に大切なのは，計算モデルとして新しいものをやっているかということと，実際に使うためにどうするかという二つを考えなければいけないだろうということです．

たとえばpetlという電総研で作られたLispは，属性リストというのをデータに書き込むことができました．確か，putとputddとがあって，要するに内部のメモリにもつか，ディスクへ書き込むことによって永続的にするというのをプログラムレベルで書いてました．

この言語は，計算モデルとしてみれば，高階関数やプログラムをデータとして扱える手続き型言語であるLispにディスクへ書き込む関数を付けて高機能化したものといえます．また，Argusなどは抽象データ型の計算モデルに，永続性やトランザクションの機能，フォールトトレランシーの機能をいれたものとみれます．

私はデータベースの研究はあまり詳しくなく，いまのところ利用者なのですが，ここまでお話した内容をもとにデータベースに対するお願いは，いまの分類にあてはまらない新たな高級性がデータベースに何かないのか，あったらぜひ見てみたいと思うことです．

あともう一つ，これは私がやりたいと思っていることは，合成するという，つまりデータベースの高機能性とプログラミング言語の高級性，たとえばなんとか計算モデルにデータベースの機能を入れたとか，プログラミング言語の高機能性に，なんとかデータモデルを単にくっつけたとかではなくて，融合した計算モデル，何か新しいソフトウェアシステムをやってみたいと思ってます．たぶん，データベースの考え方，データモデルの考え方，計算モデルの考え方にもとづいて何か作れるのではないかと，わくわくしているところです．

司会　さっきの話の追加として演繹データベー

スというのもありますよね.

布川　データモデルって計算モデルと同じくらいいっぱいあるなと思って，とりあえず歴史的な3大データモデルを取り上げてみました.

司会　多分だれも引用されていませんでしたけれども表言語あるいはスプレッドシートの永続的なものというのが考えられるでしょうね.

布川　そうかもしれません. その意味でも永続性というのは書き込めるという一つの機能ではないかなという印象をもっているんです. 計算モデル的に変わっているという話ではなくて.

司会　大堀さん，ダイナミック型というのをちょっと説明していただけますか.

大堀　アイデアは非常に簡単です. 外部のデータを扱うために dynamic という特殊な型を導入して，外部データはすべて dynamic 型をもつとします. dynamic 型の値，すなわち外部データは，データとその実際の型の組とします. この dynamic 型に対して二つの操作を用意します. 一つはデータの実際の型の検査をする操作です. もう一つは，実際の型が τ である dynamic 型のデータを静的に型 τ をもつデータに変換する操作です. これらを用いることによって，外部データを取り扱う上で必要な動的型検査を局所化して，静的な型システムの利点を保存するわけです. 重要な点は，この方式での外部データの扱いが，いままでの型理論と整合性がとれるという点です.

鶴岡　型理論で，版管理のように型が時系列で変わるような話は扱えるのでしょうか. あるクラスを使って動いていたプログラムがあり，そのクラスが変化したときに，プログラムはそのまま動くのでしょうか.

大堀　ご質問の本質は，オブジェクトがその型を変えながら進化するということを型システムでうまく表現すること，だと思います. 試みはいくつかありますが，うまく成功した例は，私の狭い知識では，ないようですね.

ポインタ是非論

堀(富士ゼロックス)　先ほど鶴岡さんから Odin の実際の経験として，データベースの操作としてポインタは使わないほうがいいという話があったと思います. これはプログラミング言語のほうからみるとどうなんでしょう. それは永続性を得るための手段だとか，それは実装だけの問題とか.

布川　僕としてはデータベースの話をするときに，ポインタという言葉が出てくる自体が驚きです，というのが答えです. だから出ないのがいいと思います.

本田(慶大)　でも Ontos とかオブジェクト指向データベースってポインタが重要ではないですか.

布川　あれは C++ なんですね.

本田(慶大)　だけどローカリティのためにポインタを使うというのが彼らの重要な宣伝文句だったと思うのですけれども，ジェムストーンの.

布川　だから使うときそうなるのではないですか. それはポインタという概念じゃなくて，ローカリティとの関係というのがあると思います.

本田(慶大)　でもオブジェクト指向にとってはポインタによるパッシングというのがポイントで，理論的にいってそれはオペレーショナルに明らかになったのはごく最近だと思うんです. だから必ずしもそう簡単には言えないと思います.

小野寺　ポインタがいけないのではなくて，Cのポインタが良くない.

鶴岡　内部的なポインタは必要だと思います.

鶴岡　Eiffel 言語を頭に置いていました.

司会　だからレファレンスみたいなのと，本当に生のアドレスみたいなのとを区別したほうがいいのではないですか.

大堀　議論の続きでポインタが必要かという点ですが，ポインタは，値の変更と，オブジェクトのシェアというこの二つの機能を実現しています. この二つの機能は，それが何と呼ばれようと，なんらかのポインタ的なものがないと表現できないと思います. これは永続であっても変わらないと思います. ですから，(堀さんの)質問へのお答えとしては，ポインタまたはオブジェクト ID みたいなもののサポートは必須だと思います.

堀(富士ゼロックス)　オブジェクト ID を実現するためにポインタというものをうまく使った. Cのポインタというのはほかに意味があるからできないということだろうと思いますが，もともと揮発データの操作と永続データの操作を同じようにしなくてはいけないという話をすると，やっぱりそのオブジェクト ID をリプリゼントするため

の揮発データのいまの表現形式はない．レファレンスという概念はCにはもっていませんね．そういうのは今後どういうふうに変更していくのでしょうか．たとえばリンクという概念を言語に持ち込むようなことが必要になるんでしょうか．

大堀 それは必要だと思いますね．たとえばレファレンス型といったものを，抽象データ型として実現し，それを型構成子として言語に導入することが必須であると思います．

牧之内 僕はポインタという概念は必要であるという気がするんです．たとえばFortranというのはポインタなんてないわけですね．科学計算しているかぎりはポインタは必要ないわけです．ところがハイレベルな言語でシステムを記述しようとすると，どうしても何かを指すということが必要になってくる．それは効率的にも必要だし，それから「何かを」指すという意味でのポインタが必要になってくるわけです．何を指しているか分からない．指したものを見て，一番上のタグを見て，ああこれはこういうものを指しているんだなということを判定することが必要になってくるわけですね．

そういう意味で，たとえばCのポインタという概念，それは必要だったわけですね．つまりCでUNIXを書くには必要な概念だったわけです．

しかしポインタという概念は何か不純である．だから1960年代の後半ですけれども，ポインタをいかにうまく説明しようという論文が結構あったわけです．僕も二，三読んだんだけれども，せいぜいレファレンスという名前を創出したぐらいであまりうまくいかなかった．

大堀 いま牧之内先生がいわれたのには二つのことがあると思います．一つは何かを指すというのと，次に，指した相手が何か分からずにダイナミックにチェックしたほうが柔軟性がある場合が多く，それが必要であるということです．この二つはある程度独立した概念であろうと思います．

2番目の概念には，その正確な構造が分からない曖昧なデータを扱う機能が含まれていますが，この機能は，残念ながら現在の静的な型システムでは実現が難しいものです．しかし不可能ということではなくて，今後曖昧なものを扱う技術ができてくれば，それをポインタと組み合わせることによって，2番目の機能をきれいに実現する可能性もあるのではないかと思います．

おわりに

司会 まだいろいろ議論があるかと思いますが，時間ですので，パネラーの方に一言ずつお言葉をいただいて終わりにしたいと思います．

小野寺 Multicsなどではデータベースと OS は融合していたと思うんですけれども，UNIXの普及でワークステーションの世界では両者は別々の道を歩むようになってしまった．というより，OSだけが一人歩きしてしまった．たぶん，そろそろデータベースの成果を採り入れるときだと思います．それもまず小さなデータベースを提供するために，データベースで現在盛んに研究されている難しい話はとりあえず置いといて，トランザクションの機能だけでもUNIXのコミュニティがうまく取り入れれば，プログラマの負担はずいぶん軽減されるんじゃないかと思います．

布川 最後の私のシートにも書いたんですけれども，データベース的な発想とプログラミング言語的な発想で，現在のどの研究会にも当てはまりそうにもないものをぜひ作ってみたいと思います．

大堀 私も同じような意見です．次世代のシステムは，データベース，言語，OS，という分類で独立に技術開発をしていてはたちうちできないほど複雑になりつつあると思われますので，各分野で蓄積した技術を統合する努力を始める必要があると思います．

鶴岡 企業の情報システムを構築している方が悩んでいるのは，やはり現実世界のモデル化だと思います．ですから，データベースの人と言語の人がもっと議論を戦わして，統一のとれたモデルを作っていくことが今後の課題だと思います．

牧之内 勝手なことを言いっぱなしでご不満な方々も大勢いらっしゃるかと思いますが今後もまた機会をみて再度討論したいものです．

司会 では，時間になりましたので，この辺で終りにしたいと思います．

どうも本日はありがとうございました．（拍手）