21世紀の情報処理への出発点

座　　長	Robert A. Kowalski	Professor, Imperial College, U.K.
パネリスト	Hervé Gallaire	G.S.I., France
	Ross Overbeek	Argonne National Laboratory, U.S.A.
	Peter Wegner	Professor, Brown University, U.S.A.
	古川　康一	ICOT, Japan
	内田　俊一	ICOT, Japan

Kowalski：このパネルディスカッションを始めるにあたり，まず第五世代コンピュータシステムプロジェクトの目的と，その目的を達成するために重要であるところの，プロジェクトで認識された技術について，簡単にまとめてみたいと思います。広い意味では，次の３つの技術分野が重要だと認識されています。

FGCSの外部にとっては，最も目立つのはアプリケーションに関する分野で，一般に「人工知能」と呼ばれる分野です。ここでは，「知識情報処理」と呼ぶ方が適当だと思います。（図１）

２番目の分野は，並列コンピュータアーキテクチャの開発です。これは，FGCSが追求してきた主要な分野の１つです。

３番目の分野は，アプリケーションとコンピュータアーキテクチャのギャップを埋めるためのソフトウェアです。FGCSのプロジェクトでは，並列プログラミング言語と制約論理プログラミング言語の開発に全力を注いできました。FGCSプロジェクトの最も独創的な点は，この３番目の分野の重要性を認識したことだと思います。

さてこのように，高度なアプリケーション，並列コンピュータアーキテクチャ，および支援ソフトウェアの各分野が，FGCSプロジェクトにふさわしい目標として確認されたわけですが，これらの分野は，21世紀のコンピュータ開発においても，やはり大きな目標になっていると思います。次に，FGCSの目標を達成する過程でどのような問題が発生したか，またその問題は将来これらの目標を達成する上で影響を与えるかどうかについて考えたいと思います。

知識情報処理の分野において，このクラスのアプリケーションの有効範囲を確認しなければ

FGCS：21世紀の情報処理に向けての跳躍

FGCS技術とは何か。

- 知識情報処理アプリケーション
- 並列プログラミング言語および制約論理プログラミング言語
- 並列コンピュータアーキテクチャ

論理プログラミング(LP)は，アプリケーションとコンピュータアーキテクチャのギャップを埋める

図１

ならないという問題があります。知識情報処理は，将来，すべてのコンピュータ処理に対する汎用的なモデルを提供するものなのでしょうか。つまり，すべてのコンピュータ処理を，知識に基づく枠組みに強制的に組み込むことができるのでしょうか。あるいは，比較的小規模な，特定分野の市場（ニッチマーケット）だけを対象としたものなのでしょうか（人工知能の市場は，現在コンピュータ業界全体の市場の約１％を占めているにすぎないと推定されています)。現在は単なるニッチマーケットですが，たとえばデータベース技術の場合と同じように，将来は大きな市場になるのでしょうか。将来のFGCS技術の有効範囲と重要性を理解するためには，以上の質問は大切なものだと思います。

では，並列コンピュータアーキテクチャに対するFGCSのアプローチを，どのように判断すべきなのでしょうか。明らかなニッチアプリケーションのために，一見特殊なコンピュータアーキテクチャを開発するのは間違いだったのでしょうか。またそれは，将来においても同じように間違いなのでしょうか。あるいは，将来，より強力な汎用コンピュータを構築するための基礎を提供する可能性があるという意味で，これは非常に独創的な発見だったのでしょうか。

アプリケーションと並列コンピュータアーキテクチャのギャップを埋めるために，論理プログラミングを選択したことについてはどうでしょうか。論理プログラミングのさまざまな形式やスタイルの間，および論理プログラミングのいわゆる「ドントノウ(don't know)」形式と「ドントケア(don't care)」形式の間，およびアルゴリズムスタイルと仕様スタイルの間には，新しい小さなギャップがいくつか発生しているように思えます。

ユーザレベルのアプリケーションに必要な論理プログラミングの「ドントノウ」形式と，並列コンピュータアーキテクチャの実行の制御に有効な「ドントケア」形式との間にギャップが存在することは，ICOTだけでなく世界中の数多くの研究機関で確認されています。ICOTでは，このギャップを埋めるための方法を研究してきました。その中で最も注目すべき方法は，モデル生成定理証明器に基づくものです。このギャップは，そのような研究成果によって埋まるのでしょうか。あるいは，将来さらに広がるのでしょうか。

最初のギャップに比べて２番目のギャップ，つまり効率的なプログラミングに必要なアルゴリズムスタイルと，プログラム仕様に有効な仕様スタイルとの間のギャップは，それほど注目されていません。この点についてはこのパネルディスカッションの後半で詳しく述べます。

最後に，過去10年間に一段と重要性が増してきた，オブジェクト指向やコネクショニズムなどの代替技術について考えたいと思います。これらの技術はFGCSの技術と互換性または補完性があるのか，または一部の批評にあるように，この２つは対立するものなのかということを判断しなければなりません。対立する場合には，これらの技術は，21世紀のコンピュータにとってFGCS技術よりも，よりよい基準となり得るのでしょうか。

それでは，パネリストの皆様を紹介したいと思います。まずHervé Gallaireさんです。Gallaireさんは，欧州の３大コンピュータメーカー，Bull,ICL,ジーメンスが出資する，ミュンヘンの欧州コンピュータ研究センター(ECRC)の最初の所長として有名です。現在は，3,500人以上の従業員を抱えるフランスの大手ソフトウェア企業GSIにおいて，計画の立案とソフトウェアのインプリメンテーションを担当しておられます。また，国際論理プログラミング協会(International Association for Logic Programming)の理事長も兼ねておられます。

次のパネリストは，アメリカ合衆国のブラウ

ン大学のPeter Wegner教授です。Wegner教授は，やや異なる思想学派に所属しておられます。ソフトウェアの主流となっているアプローチに関して，多大な経験を積んでおられます。ALGOLとADAに対して重要な貢献をされ，最近ではオブジェクト指向プログラミングの分野で業績を挙げておられます。このようなテーマに関して，10冊以上の本を著作または編集されました。特に論理プログラミングとオブジェクト指向プログラミングに関する論文集を編集されています。

次のパネリストのRoss Overbeekさんは，アメリカ合衆国のアルゴンヌ国立研究所に所属しておられます。Overbeekさんは，FGCS技術の多種多様な分野で顕著な業績を残されています。特に，自動定理証明の分野に大きく貢献されました。この分野では，アルゴンヌ国立研究所の彼の研究チームが，これまでのところ世界をリードしています。また，並列コンピュータアーキテクチャの開発にもかなり力を入れてこられました。ごく最近は，研究の方向を大幅に軌道修正し，一般的なアプリケーションに焦点を当て，特に遺伝子配列アプリケーションを重視されるようになりました。

次のパネリストの古川康一さんは，電子技術総合研究所(ETL)の研究員としてFGCSプロジェクトの計画および立上げを担当されました。古川さんはこのプロジェクトの開始当初からICOTで研究を行い，まず最初はソフトウェアとFGCS核言語の研究を担当する第2研究室の室長を務められ，現在はICOT研究センターの次長です。ICOTで科学分野の管理を担当されているだけでなく，FGCSソフトウェアのあらゆる面に関して数多くの研究文献を発表されています。古川さんの研究上の判断は，FGCSプロジェクトの研究の方向付けに大きなインパクトを持っています。

次のパネリストの内田俊一さんも，同僚の古川さんと同じように，かつてETLで研究に従事されていました。そこで，ICOTへ入る前からFGCSプロジェクトの計画立案を支援しておられました。ICOTへ移ってからは，まずコンピュータアーキテクチャの研究を担当する第1研究室の室長代理を務められ，現在はICOT研究センターの研究部長の職にあります。内田さんは，PSIマシンとPIMアーキテクチャの開発を担当されました。彼は，ICOTの後継プロジェクトのリーダーになる人物だとのもっぱらの噂です。

私自身もこのパネルに参加いたします。それでは，最初のパネリストであるHervé Gallaireさんにプレゼンテーションをお願いいたします。

Gallaire：こんにちは。本日私を招待してくださったプログラム委員の皆様に感謝したいと思います。この場で論理プログラミングの将来について議論することは，私にとって大きな喜びであるとともに名誉なことだと思います。1979年当時はまだICOTが結成されていなかったので，今回よりはるかに聴衆の少ないETL会議に出席したことを覚えています。とはいえ，当時，渕さんと古川さんはすでにリーダー的な役割を果していました。その時，私たちは論理プログラミングとLISPの長所について議論して比較を行いました。2人はその後で決定を下したわけですが，その決定は正しかったと思います。1979年の小規模な会議に比べると，本日この場で意見を述べるのは，私にとって非常に難しいことです。しかし，Bobが触れた問題の一部について述べてみようと思います。

私たちは，ここで，21世紀の情報処理に向けての跳躍台として論理プログラミングについて論じています。さて，このスライド（図2）は，この疑問に対する私の答になっています。この内容について，ごく手短にお話します。

私が，「LPは情報処理の跳躍台にはならない」

21世紀の情報処理に向けての跳躍台

- Logic Programming (LP)は跳躍台であり得るか…
  - あまりそうではない
  - 私たちは皆無駄な研究をしてきたのか
  - LPとは何か，何に適合するのか
- 別の跳躍台はあるか
  - 多数の跳躍台がある

図2

と言うとき，それは何を意味しているのでしょうか。Bobがその疑問をどんなに正確に分析しようと努めても，それはやはりごく一般的な疑問であり，それに対する単純な回答は存在しないと思います。つまり，私たちが行うべきことは，論理プログラミングを単に21世紀の情報処理に向けての一般的な跳躍台と見なすのではなく，論理プログラミングがどの点で情報と知識処理を助けてくれるかを見つけ出すことであると思います。

「私たちは皆，無駄な研究をしているのか」という疑問に対しては，明確に「いいえ」と答えることができます。皆さんが今回の会議でご覧になった研究成果や，他で行われている研究を見れば，答が「いいえ」であることは明らかです。では，その答が「いいえ」であるとしたら，私たちはどこに位置付けられ，またどこにいるべきなのでしょうか。そして，将来はどこに行こうとしているのでしょうか。この点が一番お話ししたいことですが，私の考えでは，論理プログラミング以外にも単一で存在する跳躍台はないということを明確にしたいと思います。

跳躍台としての論理プログラミングの役割を議論するとき，それは実際には何を意味しているのでしょうか。ここでスライド(図3）を使って，情報処理を全体として見るとなぜ戸惑ってしまうのか理由を説明したいと思います。ただし，その点をうまく説明することはできないと思います。しかし，時間をかけて細部にわたって議論すれば，一般論に終止することなく，結論を導き出すことができると思います。

メーカー，研究者，ユーザなどは，システムに対して異なった視点を持っています。各々さまざまな要求を持っており，論理プログラミングがその要求のすべてに適合しなくても仕方がありません。また，論理プログラミングを使用することにしたとしても，はたして世界のモデル化や推論のために使用できるのか，または論理プログラミングが世界そのものになることができるのでしょうか。これでは，質問がさらに複雑になります。（図4）現在は，システムを階層構造として構築しています。したがって，その各視点から議論することによって，どの階層で論理プログラミングが現われるか研究する価値があると思います。このような複雑な状況を分析してみると，Alan Robinsonが月曜日の講演で抽象計算機械としてのチューリングマシンについて論じたときに使用した意味を除いて，論理プログラミングを始めとして，どのプログラミングパラダイムも基盤とはなり得ないと思います。チューリングマシンの意味では，論理プログラミングは，基盤となるパラダイムと言うことができますが，ただし，この安全な領域を出るやいなや，他のものを見ずにはいられな

実際には何を意味しているのか

- 情報処理の概念は多様な状況への対応が可能
  - 全体としての計算
  - 具体的な分野
- 質問の背景にいるのは誰か
  - 研究者
  - メーカー
  - アプリケーション開発者
  - システム開発者
  - アーキテクチャエンジニア
- 実際のものとして，またはモデルとして
  - 計算の研究
  - アクターの研究
  - 計算
  - 情報システム

図3

グローバルな計算の視点

- 世界に対する見方の階層構造
- 1つのレベルに対する複数の見方
- 各種のツールが必要
- さまざまな抽象化が必要
- LPを始めとして，どのプログラミングパラダイムも絶対的な基礎にはならない
- しかし，限界まで押し進めようとすることが必要（例：ハードウェア開発は名案だった）

図4

くなります。

要点は，先ほどご紹介した概念を本当に理解したいならば，たとえば論理プログラミングがどこに適合するか見るには，1つの方向を選んで，選択したパラダイムを限界まで押し進めざるを得ないのです。実際に，論理プログラミングをある意味の統一パラダイムとして見なす人々は，その周りの壁を押し続けてきたわけであり，また現在でも壁を押し続けています。そして，そのパラダイムについてだけでなく，他のパラダイムについても研究を進めています。実際に理解することができ，また進歩に大きく貢献するのは，まさに境界を超えて壁をさらに押そうとするときです。具体的な例を挙げてみましょう。論理プログラミングで並行性の問題に取り組むことによって，一般的な並行性についてかなりの理解が得られました。また，並行性の実現にあたって最終的にどのパラダイムを

モデルと研究の視点

- 一貫した見方を提供してくれる主な候補
- つなぎ目がわからないように世界の各部分をうまく総合化
- 主な貢献内容：ルールベースの演繹的データベース管理システム，問題解決と人工知能，時間と信念に関する具体的な論理プログラミング，否定の処理，並列プログラミング，制約プログラミング
- 場合によって，優勢な位置を占める
- 誰もが採用する単一の包括的なモデルにはならない
- その理由は，人間の知能が膨大すぎるからである
- また抽象化の視点が多すぎるからである

図5

使用するとしても，並行性を理解する上で前進できたと思います。

さて，この質問をモデルと研究の視点から見てみましょう。（図5）1つ前のスライドのところで述べたように，私の答は「はい」です。明らかに，論理プログラミングは，世界に対する一貫した視点を提供する主要な候補です。この視点からは，重要な反論はないと思います。すべてのことを簡単かつ容易に解決できることを意味している訳ではありませんが，いくつかの分野で進歩が見られたと思います。ここでは，ほんの少し例を挙げるだけで，決して包括的なものではありません。並行性や言語の分野に対する貢献に比べてあまり知られていないかもしれませんが，演繹データベースの世界に対して論理プログラミングが果たした貢献を強調したいと思います。データベースの将来は、長年にわたって学んできたことから考えると，他の何よりも論理プログラミングの将来に関わっていることは間違いないと思います。

同様に，問題解決の将来は，論理プログラミングの将来に関わっています。制約言語は，私たちが見つけ出すことができた解決法の好例であり、今後も長期にわたって確実に多用されると思います。これこそ，私たちの分野だけの問題解決や記号計算だけでなく，数値と記号の計算の結合など，この特殊なアプローチを通して実現可能になる数多くの困難な問題に対して取るべき道です。ここで，並行(concurrent)，並列(parallel)プログラミングなど，すでに述べた例を挙げます。

場合によっては，論理プログラミングはすでに有力な地位を確保しています。言うまでもなく，それが至る所で見られるわけではありません。確かに，いくつかの領域では論理プログラミングの地位を改善できますが，すべての分野で論理プログラミングが優勢を誇ることはあり得ないでしょう。人間の知能はあまりにも膨大であり，物事に対する1つの見方がすべてに於いて優勢を占めることはあり得ないと思います。万能計算機構は，この原則をよく表しています。もう1つ指摘したい点は，問題を解決するとき，人間はその問題空間に対してさまざまな抽象レベルを考え出すのに非常に優れているということです。人間は，適切な抽象レベルで好んで仕事をします。1つのレベルで表された解を，別のレベルに写像するのが有用なことがあります。また，逆にそれが危険な場合もあります。ところで，論理プログラミングにとっては，すべてのものを論理プログラミングに写像するのは危

険です。適切なレベルでの推論が必要です。

それでは，商業的な視点に目を向けてみましょう。（図6）論理プログラミングの将来に関する質問に対する解答は，他の視点よりも商業的な視点から考えた方が簡単です。答は「いいえ」です。私たちはまだ，到達したいと望んでいる位置には至っていません。将来その位置に到達するかどうかは，21世紀ではなく，それよりも近い将来においてこの問題をどのように処理するかに大きく依存しています。21世紀に入る前に，この点を心配しなければならないと思います。21世紀になってからでは遅すぎると思います。新しい考えに対しては，反対意見がつきものです。そのため，商業的な分野ではあまりうまく行きませんでした。ただし，それだけではないと思います。論理プログラミングの長所を説明し実際に証明しようとすればできたはずなのに，その努力を怠ったのではないかと思います。世界，特に商業の世界は一時的な流行に支配されています。たとえばオブジェクトプログラミングは，商業の世界では間違いなく一時的な流行です。とはいえ，オブジェクトプログラミングには，一時的な流行以上の何かがあることが理解されています。オブジェクトプログラミングは，商業の世界で理解できるものを提示しています。それとは対照的に，私たちは，論理プログラミングによって具体的にどんな利点が得られるのかをはっきりとは説明していません。一例を挙げるとすれば，演繹推論データベースを，SQLの世界やCOBOLに十分に関連付けていません。私たちには，もっと出来ることがあると思います。私たちにできることの例を挙げてみたいと思います。給与計算アプリケーションの例を考えてみましょう。論理プログラミングのパッケージとして給与計算プログラムを作成すれば，非常にうまく行くと思います。このパッケージは，ひとりひとりの従業員に対応して並列ストリームを実行する並行論理アプリケーションとして作成できると思います。また，規則を宣言的に示したい場合には，コンピュータ側で制約プログラミングを使用することもできます。この場合にはデータフローが必要になり，制約プログラミングを通して提供されます。また，制約プログラミングをスケジュー

商業的な視点

- まだ希望の段階にまで達していない
- ニッチ（例：ツールへの埋込み）
- 新しい考えに対する抵抗
- 研究が決定的でない場合が多い：リスクがない
- 一時的な流行があふれている
- オブジェクト：一時的な流行……しかしそれ以上のもの

- 論理プログラミングの長所は，オブジェクトの長所のようには説明されていない
  - 現実の世界を通してCOBOLとSQLに関連付ける
  - ビジネスルールと総合チェックの長所を説明する
  - 既存のソフトウェアとの総合化
  - 市場の開発（例：制約）
  - 拡張機能の埋込みまたは拡張機能の開発（例：制約）

図6

ル作成や職務割当てなどをサポートする人的資源パッケージに結合すれば，制約プログラミングの使用によって利益を得ることができます。商業の世界で理解できるのは，このような成果だと思います。私たちはそのような利点を説明しなければなりませんが，説得力を持ってこのような例を説明できるようになるまでには時間がかかるでしょう。

ソフトウェアの分野でさえ，私たちは0から構築しています。たとえば，論理ベースの計算エンジンから制約プログラミングを構築しています。その代わりに，論理プログラミングの世界をすべて忘れるか，または忘れるように努力して，純粋なC言語などの既存のエンジンと結合することによってこれを構築すべきではないかという疑問がわいてきます。このようなことが完全にできるとは思いませんし，また現在でも論理プログラミングからあらゆる恩恵を受けています。それでも，両者を合わせた解決法を無視すべきではないと思います。

結論としては，成功の可能性を認識し，それを目指して研究に励む必要があると申し上げたいのです（図7）。ここまで，3つの分野，つまり，まだすべてが開始時期にあり，あまり安定しておらず，したがって私たちが強い影響を与えることができる3つの分野についてだけ述べてきました。制約プログラミング言語をリストアップして，すでに例も取り上げました。CASEツールをリストアップしました。これは，まだ成熟していない世界です。知識表現要求は，演繹推論技術とともに本質的であり，論理プログラミングで達成可能な種類のモデル化には大きな可能性があります。ここで挙げた3番目の例は，「ワークフロー言語」の例です。このような言語をコーディネーション言語と呼ぶ人もいます。私が考えていることを述べたいと思います。コンピュータの世界はさまざまな分野に広がって行きます。それは，意志決定ベースまたは計算集約ベースだけでなく，もっと多くの分野で使用されるようになります。通信としての計算処理が開始されており，それはますます普及していくでしょう。クライアントのサーバアーキテクチャまたは協調アーキテクチャから，数多くの技術が実用化されつつあります。しかし，現在，これらの技術では通信とプロセスとの同期化の問題は処理できません。ここに取り上げた製品にみられるのは，低レベルの基礎的な技術です。私たちに必要なのは，式典の司会者のような一段と有力な技術なのです。この分散世界を組織化するには指揮者が必要です。こ

研究所の内外を問わず精力的な研究が必要

- 成功の可能性を見極め，それを目指す
- 論理ベース制約プログラミング言語
- CASEツール
- ワークフロー言語
  - アプリケーションは分散していたり，していなかったりする通信オブジェクトから作成される
  - それらを連結する技術が存在する：AppleEvents,OLE,CORBA,ToolTalkなど
  - LP（および特にCLP）は必要とされる偉大な指揮者になりうる
- 商用に耐え得る実装に取り組み，現実のアプリケーションの問題を処理する
- より大きなリスクを負う

図7

れこそ, CLPのすべての成果を活用できる分野であると思います。この場合には, CLPは，何らかの形式の制約論理プログラミングまたは並行論理プログラミングを意味しています。これは，優れた仕事をするための能力を十分に備えています。これで，私の発表を終わりたいと思います。ありがとうございました。

Kowalski：次のスピーカに移りたいと思います。討論の時間は最後に設けることにしましょう。ではPeter Wegnerさん，どうぞ。

Wegner：こんにちは。このパネルに参加させて頂いてたいへん光栄です。特に，私の専門は論理プログラミングではないのに唯一の部外者として招待いただいたことは，とても名誉なことだと思います。部外者の役目として，どうしても挑発的にならざるを得ません。私は，問題解決における論理プログラミングについて，物議をかもすような立場を取りたいと思います。つまり,「21世紀のアプリケーションプログラミングでは，オブジェクト指向プログラミングが論理ベースの推論パラダイムよりも優勢な地位を占める」という意見を述べたいと思います。

しかし，議論に入る前に, ICOTの絶大な影響についていくつかお話しさせていただきます。

ICOTは，論理プログラミング，並行論理プログラミング，制約論理プログラミング，並列マシンのアーキテクチャ，並列マシンのためのシステムソフトウェア，および遺伝子に関するアプリケーションなどの応用分野に対して，多くの科学的な貢献を行いました。ICOTは，日本の研究基盤を築くとともに，論理プログラミングの研究を世界各国に普及する上で貢献しました。また ICOTは，共同研究の新しいモデルを提供しました。たとえば, Esprit研究プロジェクトはこれに触発されたものであり，欧州経済共同体に大きな刺激を与えました。さらに，新しい世代の若い研究者に対して活気に満ちた研究環境を提供し，またReal World Computing(RWC)プロジェクトに移転できる基盤を提供しました。ICOTは，その多大な科学的，社会的，および政治的な効果という点において非常に大きな成功を納めました。

モデル化パラダイムと推論パラダイムは相補的なものでしょうか。それとも，その間には必然的なトレードオフが存在するものなのでしょうか。ここで私が主張したいのは,「モデル化パラダイムと推論パラダイムは，互いにトレードオフが存在するいう意味で非互換的である」ということと,「21世紀にはモデル化パラダイムが推論パラダイムよりも優位に立つ」という２つの点です。論理プログラミングは推論パラダイムの典型的な例であり，またオブジェクト指向プログラミングはモデル化パラダイムの典型的な例となります。推論とモデル化の関係は, ICOTの第五世代コンピュータプロジェクトとその後継プロジェクトとして提案されているRWCプロジェクトの関係と同じです。第五世代における計算は推論による問題解決でしたが, RWCPにおける計算は，モデル化による問題解決です。

モデル化と推論の論争は，合理主義と経験主義の2000年にわたる哲学的な論争を計算の分野で受け継いだものです。

デカルトの「我思う故に我あり」という言葉は，思考が知識の基礎であるという合理主義者の信条の真髄です。彼はデカルト幾何学を提唱し，幾何学を代数に還元しました。これにより，物理学を数学に還元できるという可能性，および論理に対する計算の可能性を信じる傾向が強くなりました。ヒュームは,「演繹的な論理は，帰納的な法則や偶然の法則の基礎としては不十分であり，従って経験主義の基礎となり得るのは物理学のような経験的科学である」と主張しています。ヒュームに影響されて，カントなどの哲学者は，合理主義の信条を純粋理性によって再検証しました。しかし，ヘーゲルは，それ

に影響されることなく，ラッセルなどの数学哲学者に強い影響を与えることとなった欠陥(flawed)のある推論形式（弁証法的推論）に基づいて超合理主義を採用しました。

20世紀の前半は数学的合理主義の時代でした。ラッセルは，数学の原理(Principia Mathematica)において数学を論理に還元しようと試みました。また，ヒルベルトは，すべての数学的な問題を機械的な数学の規則で解決しようと計画しました。しかしながら，1920年代の終わりから1930年代にかけて，ゲーデルの不完全性，チューリングの計算不能性，およびハイデルベルグの量子論の領域における不確実性理論などのいくつかの研究の成果によって，このような合理主義は実現不可能であることが明らかになりました。その結果，物理学における操作主義，心理学における行動主義，および今日まで続く数学における形式主義からの離脱など，経験主義への回帰が発生しました。

オブジェクトプログラミングは経験主義者の試みですが，論理プログラミングは合理主義者の発想であり，かつ合理主義者によって制約されています。論理プログラミングは，計算に対する入力として観察されたデータの必要性を否定するものではありませんが，いったんデータの収集が完了すると，演繹的論理の法則に従ってすべての計算が実行されることを意味しています。論理プログラミングは，私たちの能力をアプリケーションの直接的なモデル化に限定する制約された計算の形式であり，またオブジェクト指向プログラミングに代表される経験主義者の影響は，私たちをこの合理主義者の制約から解き放ってくれるものです。

計算について，状態遷移パラダイム，オブジェクト指向モデル化パラダイム，Prologに基づく推論パラダイムの３つのパラダイムを考えてみましょう。状態遷移パラダイムは，手続き指向のパラダイムです。プログラムはアクションの列であり，計算ステップは状態の遷移になります。モデル化パラダイムの例としては，オブジェクト指向パラダイムがあります。プログラムは相互作用を行うオブジェクトの集合であり，計算ステップはメッセージ交換に当たります。推論パラダイムの例としては，論理パラダイムがあります。プログラムは推論のルールの集合と事実のデータベースであり，計算ステップは論理的な推論に当たります。

この会議では論理パラダイムはかなりよく理解されていますが，ここでモデル化パラダイムについても多少述べたいと思います。オブジェクト指向パラダイムは，実体をその観察可能な振舞いによってモデル化するという点で，科学的なモデル化を模倣しています。オブジェクトは，物事を，その観察可能な属性，つまり相互作用を決定する手続き，および各オブジェクトに対して意味を持つメッセージの集合によってモデル化します。その過程で，“what”を潜在的なすべての“how”で指定し，潜在的なすべての振舞いによって宣言型の仕様を提供します。従って，オブジェクト指向パラダイムは，計算の領域でモデル化の異なったオブジェクトにまたがる本質をとらえます。

Bob Kowalskiの分割統治(divide−and−conquer)というアフォリズム「アルゴリズム＝論理＋制御」を，以下のように変更してみます。

プログラム＝宣言性＋制御＝“what”＋“how”

宣言性と論理プログラミングの論理を同一視するのであれば，この変更は意味がありません。しかし，宣言性の定義を拡張して，どうやって計算するかとは独立に何を計算するかを指定する方法を包含するのであれば，この変更が意味を持つようになります。オブジェクト指向プログラミングでは，特定の計算（アクション）とは独立にオブジェクト（物事）を定義し，した

がって宣言的であると言われます。オブジェクト指向プログラミングは，オブジェクトに対する特定の効果を確定(commitment)することなく，すべての可能な影響という点からオブジェクトを指定します。オブジェクトの宣言性は，論理プログラミングの数学的な宣言性とは非常に異なります。しかし，科学的なモデル化においてテーブルなどのものを指定する方法と非常によく似ているという点で，ある意味ではより有効です。

オブジェクトの制御メカニズムは，論理プログラミングの制御メカニズムとはかなり異なります。オブジェクト推論における制御は，以下の形式のselect文で指定できます。

select(op1, op2, ..., opN)

ここで, op1, op2, ..., opNは，オブジェクトのオペレーションまたはメソッドです。このselect文はオブジェクト内では暗黙的ですが，AdaやCSPなどの並行オブジェクトや並行プロセスでは明示的となり，Dijkstraのガードコマンドの１つとなります：select(G1 | op1, G2 | op2, …, GN | opN) select文は，本質的には「読込み－評価－印刷」のループであり，メッセージを待ってそれを実行します。オブジェクトは，ローカルには非決定論的であり，次に何を実行するか知りません。入力を待って待機しており，入力が到着するとそれを確定(commit)します。したがって，これは，非決定性のコミッティッド・チョイス形式です。「オブジェクトインタフェースにおける制御はコミッティッド・チョイスselect文で実現される」という概念は，しばしば並列オブジェクトの属性と考えられますが，しかしこれは順次オブジェクトにも当てはまります。

次に，モデル化における開放型システムの役割について考えてみたいと思います。「開放型システム」という用語は，増分的(incremental)，拡張可能(scalable)，変更可能(modifiable)で，かつ進化(eveloving)するシステムを意味しています。開放性は，論理では獲得できないモデル化の属性です。増分的な変更には２種類あります。１つは，刺激に対して動的に応答するリアクティブネスです。もう１つは，プログラマによるモジュール化システムの静的な拡張(extensiblility)を容易にするカプセル化です。リアクティブネスでは，生物学的な進化におけるのと同じような組織の時間的な進化を獲得できます。一方，カプセル化では，ソフトウェア工学におけるのと同じように，外部的な変更を獲得できます。

開放型システム＝カプセル化＋リアクティブ
＝空間的な拡張性＋時間的な進化

システムは，静的かつ動的に変更可能な場合に，強度に開放的であるといいます。オブジェクトは，本質的に強度に開放的です。つまり，環境に対して反作用し，またモジュール化されていて簡単に拡張できるという意味でリアクティブです。論理プログラムは，強度に開放的ではありません。つまり，これから述べるように，技術的な意味でリアクティブではありません。節を追加したり，またデータベースに事実を追加したりすることにより静的には拡張できますが，オブジェクト指向システムほど直接的にアプリケーション領域をモデル化することはできません。その結果，アプリケーション領域の小さな変更が，必ずしもプログラムの小さな変更に変換されるとは限りません。

次に，なぜ論理プログラムがリアクティブであり得ないかについてお話します。リアクティブなシステムは，刺激に対する反作用を取り消せないという意味で，環境に対する取り消し不

能な副作用によって計算を行います。純粋な論理プログラムは，定理が証明されるまで反作用を持つことができません。つまり，その結果が真であり，反作用の可能性がないと確認できるまで，取消不能な確定を行うことができません。リトラクティブネスと呼ばれる属性，つまり物事が全く発生しなかったかのように取り消すことができる属性は，リアクティブネスとは互換性がありません。

並行論理プログラミングは，コミッティッド・チョイス非決定性(committed choice nondeterminism)のために，論理的な完全性にとって本質的なリトラクティブネスを犠牲にしています。リアクティブネスを実現するために論理的な完全性を放棄して，オブジェクト指向プログラミングの制御構造と非常によく似た制御構造を採用しました。これについて，節の頭部に同じ述語Pを持つ並行論理プログラムのすべての節の集合を考えてみます。

P(E1):- G1 | B1 - head P(Ei) , guard Gi, body Bi
P(E2):- G2 | B2 - reducible if unifies(E, Ei)and Gi
...
P(En):- Gn | Bn - irrevocable committed choice

これは，ゴールP(E)が証明されたときに潜在的に起動可能なすべての節の集合に対応します。純粋な論理プログラムでは，ゴールが証明されるか，またはすべての代替節が網羅されるまで，すべての節が試行されます。すべてのものが最終的に試行されるまで，システムはどの代替節が正しい解につながるかを知らなくてもよいため，このすべての代替節に対する網羅的な探索をドントノウ非決定性(don't-know nondeterminism)と呼びます。ドントノウ非決定性は，しばしば，バックトラッキング，およびリアクティブネスに対する矛盾によってインプリメントされます。並行論理プログラミングシステムは，ドントノウ非決定性を取り扱うことができません。そして，確定の効果を無効にする要求がもはや存在しないため，リアクティブネスを許すコミッティッド・チョイス（ドントケア）非決定性を選択しています。その結果，与えられたゴールP(E)を満足する場合に代替節を選択するための制御構造は，オブジェクトのメソッドの選択に正確に類似しています。指定された述語Pの節の集合は，オブジェクトのメソッドの集合に似ています。

推論対モデル化の設計空間を２つの次元で考えることで締めくくりたいと思います。１つは，宣言型または"what"の次元であり，推論対モデル化に対応しています。もう１つは，命令型または"how"の次元であり，リトラクティブ対リアクティブに対応しています。純粋な論理パラダイムはリトラクティブな推論パラダイムであり，またオブジェクト指向パラダイムはリアクティブなモデル化パラダイムです。

純粋な論理プログラミング(LP)→推論＋リトラクティブ
オブジェクト指向プログラミング(OOP)→モデル化＋リアクティブ

"what"と"how"の仕様は２行２列の表となるため，２つの可能性を考えなければなりません。（図８）.

What \ How	Retractive	Reactive	
Reasoning	Pure LP	CLP	Shared Constraints
Modeling	DLC	OOP	Distributed Components
	Complete	Flexible	

図８：Design Space for What and How Specifications

並行論理プログラミング(CLP)=推論＋リアクティブ

分散論理プログラミング(DLC)=モデル化＋リトラクティブ

「推論＋リアクティブ」の組合せは，並行論理プログラミングに対応します。「モデル化＋リトラクティブ」の組合せは，現在は，十分に定義されたいずれの言語にも対応していません。しかし，ローカル「ドントノウ」非決定性をカプセル化する分散論理コンポーネントで，かつコミッティッド・チョイスインタフェースを持っていないものについて研究が進められています。

リトラクティブな推論とリアクティブなモデル化の組合せは，他の２つの組合せよりも安定しているように見えます。リアクティブな推論とリトラクティブなモデル化システムは，もう１つの混合型の設計方法です。並行論理プログラマは，リアクティブな推論は２つの世界の最もよいところを組み合わせたものだと主張しています。しかし，他の言語パラダイムの研究者は，このアプローチは２つの土台の間に落ち込んでしまったものであり，論理的でもまたリアクティブでもなくなってしまったと感じています。分散型の論理コンポーネントは，リアクティブな推論システムに比べてあまり開発が進んでいません。分散型の局所性と完全性を組み合わせることには利益があると主張することもできます。しかし，リアクティブでも推論システムでもなくなるため，ローカルな完全性は不適切な混合型であることがだんだん分かってくるのではないでしょうか。

結論として，モデル化パラダイムは，異なる宣言型の概念に基づいており，またリアクティブであると言う意味で，推論パラダイムとは根本的に異なっています。オブジェクト試行型のリアクティブネスと論理的な完全性の間にはトレードオフが存在し，そのどちらかを選択しなければなりません。大規模なプログラミングでは，論理的な完全性よりもリアクティブネスの方が重要です。したがって，21世紀には，大規模なプログラミングにおいては，論理パラダイムよりもオブジェクト指向パラダイムの方が優勢になると思います。

Kowalski：ありがとうございました。次の講演者はRoss Overbeekさんです。

Overbeek：私の視点は，おそらく他の講演者の方々とは少し違うと思います。私たちに与えられた課題は，21世紀における論理プログラミングと並列処理の役割について述べることです。

私たちが直面すべきことについてお話しし，次に本質的な質問を提示したいと思います。私たちは，心の中で次の問いかけを行わなければなりません。すなわち，「最良のアプリケーション，つまり特定のアプリケーションの最良のインプリメンテーションを，論理プログラミングシステムに基づいて作成できるのはどういう場合か。そして限定されたインスタンスだけでなく，それ以外の分野でもそれが実現されるのはいつか」という質問です。

この技術の役割を理解するには，このような本質的かつ基本的な質問に目を向けなければならないと思います。これらの質問は，私たちがその価値を認識し好んで論議する基礎的な研究テーマから生まれてきたものです。

私の限られた視点から申し上げますと，技術はいくつかの発展段階を経て発達していくと思います。

1．基礎的な問題に対する早期の探求と公式化
2．新しい技術の使用可能性に対する認識を深めること
3．「単純な核」となるプロトタイプ的な問題に関する実験

4．いくつかのアプリケーションで徐々にその技術が採用されていく段階
5．技術が広く受け入れられる段階

私は，早くから並列アプリケーションに取り組んできました。そして，並列アーキテクチャ（およびそのアプリケーション）がこの各発展段階を経て行くのを見てきました。「並列マシンについて脅威を感じる必要は何もありません。やがて無用の長物となり，逐次マシンに駆逐されるからです」と言われたことをはっきりと覚えています。

並列処理は，５番目の発展段階に入ったところであり，これから広く採用されると信じています。逆説的ではありますが，「並列処理が最終的に受け入れられたときには，それはかつて考えていたほど重要ではないだろう」と思うようになりました。しかし，今は論理プログラミングの方に関心が傾いています。なぜならば，そちらの方がより重要なテーマだとわかったからです。しかし，21世紀に論理プログラミングが果たす役割については，まだ明らかになっていないと思います。その役割はまだ決まっていないし，また技術的な側面から決めることもできないと思います。

論理プログラミングという技術の将来をある程度予測するために，現在登場しつつあるアプリケーションを検討することから始めてはどうかと思います。私の考えでは，それらのアプリケーションは３つの基本的なカテゴリに分類されます。

ａ）まず，私が述べたように，表現力に基づくアプリケーションの集合があります。もちろん，まだ十分に説明した訳ではありませんが，しかしその意味するところは，プロトタイピング，データベースの利用，およびインタフェースの作成が容易であるという理由でPrologのような言語を使用する，一群のアプリケーションがあるということです。

ｂ）２番目に分散型の計算に基づくアプリケーションのカテゴリがあります。その最も良い例は，電話の交換をシミュレートする作業です。

ｃ）最後に，私がより力を入れてきた分野は，並列アプリケーションの協調問題です。この分野では，コミッティッド・チョイス論理プログラミングの基本的な考え方が，広範囲にわたるアプリケーションで受け入れられるのを見てきました。実際，現在の時点では，Intel社のDeltaをかなり強力なマシンの１つだと考えています。そして，そのマシンに対する最初のアプリケーションは，KL1言語と同じ知的な基盤を持つPCN言語によるものだと思います。

このように異なる分野があります。これから直ちに引き出せる驚くべき結論は，Bobが取り上げたように，一般的な枠組みとしての論理以外には統一化の原則は存在しないということであり，これらのカテゴリは，実際に大幅に異なるアプリケーションのクラスであるということです。

「統一化は発生しますか」と問うことができます。これは，言い換えるとPrologなどの言語とコミッティッド・チョイスに基づく言語との間に統一化が行われるかという質問です。１つの答として，次のものが考えられます。「いいえ。本質的には，PrologやUNIX，Cワークステーションなどのアプリケーションを駆動する技術があり，それはソフトウェア市場の大半を占めています。その結果，コミッティッド・チョイスアプリケーションは，主として数値計算を行う超大型のマシンに限定されてしまうでしょう。」やや異端的な意見かもしれませんが，統一化を試みないでこの２つの環境を構築すべき

だと思います。

もっと客観的な見方があると思います。リーダーの中には，まずある分野を徹底的に理解し，適切な解決法を考え出してからそれを適用すれば失敗するはずがないと考えている人もいます。残念ながら，このような考え方は昔も今も私の視点ではありません。しかし，この見解は理解できますし，この見解を待っているグループが存在する理由もわかります。

しばらくの間だけでも，この技術を使用して大きなアプリケーションを開発している人の視点に立ってみてはどうでしょうか。つまり，アプリケーションを開発する人の視点から世界を考えるように，自分自身を仕向けてはどうかということです。これは，技術の向上という視点ではなく，与えられたアプリケーションが何であろうと，その最良のシステムを迅速に開発する人の視点です。論理プログラミングの場合には，次の２つの基本的な要因の間にトレードオフ関係があることがすぐにわかると思います。

1．１つの要因は，論理プログラミングの技術を使用する市場の規模です。幅広く利用できない技術を市場に導入すると，製品市場が縮小するという問題に直面します。
2．一方で，開発コストの要因があります。これは，多くの点で論理プログラミングの長所となります。宣言型の論理と表現の容易さに基づいて，アプリケーションの開発コストを削減できます。

しかし，このトレードオフ関係を理解するには，製品開発に従事する者なら誰でもぶつかる諸問題に目を向けなければなりません。私の時代にプログラミングで提案，開発，および使用されてきたコンピュータ言語の一覧表を作成するとしたら，どういうことになるのでしょうか。計算機科学者がその歴史を認識していないとしたら，つまり言語の特徴だけに頼って結論を下した場合，何人の科学者がC言語を重要な言語として取り上げるでしょうか。

C言語は，多くの要因によってその重要性が認められていると思います。その要因の多くは，プログラミング言語としての内在的な特徴には全く関係ありません。同様にUNIXについても考えることができます。なぜUNIXが重要になったのでしょうか。現在のUNIXの流行をもたらした要因は何だったのでしょうか。さらに不思議なのは，現在MS-DOSが世界中で主要な役割を果しているのはなぜなのでしょうか。21世紀の論理プログラミングの役割を理解するためには，以上の質問について考えていかなければならないと思います。技術の採用と普及を決定する要因は，計算機科学者が通常考える要因とは全く異なっています。

論理プログラミングの使用については非常に楽観的に考えているので，次に私が取り組んでいるアプリケーションの議論に移りたいと思います。実際のところ，現在は，大半の研究時間を生物学のアプリケーションに費やしています。重要なアプリケーション分野の１つに，遺伝学的な順序データに関連する統合化データベースがあります。そのようなデータベースが巨大化かつ複雑化するのは必至であり，今後何十年間にわたって誕生してくる技術を歪めてしまうと言う人々もいますが，必ずしもそうとは言えないと思います。そのようなデータベースに関して本当に重要なのは，複雑な構造を持っていること，統合化が困難なこと，および「大量のデータを機能的な資源に変えるには，まさに論理プログラミングと論理の長所が必要なこと」だと思います。実際のところ，最良の統合化は，日本，欧州，アメリカでの論理プログラミングの研究努力から生まれると思います。しかも，この点は，生物学界では正しい統合化のための最も有望なアプローチの１つと認識されています。

生物学的データベースのような分野については，非常に楽観的に考えています。他にも，そのような分野を探すべきだと思います。

論理プログラミングパラダイムのもう１つの側面を反映する分野としては，神戸大学の松田秀雄氏およびイリノイ大学の研究グループと共同で行った系統学の研究があります。系統学では，進化の系統図の再構築を行います。すなわち，ある意味では祖先の系統を再生成し，進化の過程を遡り，万物の祖先に対する見方を確立しようとします。このアプリケーションでは，性能が重大な意味を持ちます。そのアプリケーションをワークステーションで実行する場合，最大尤度に基づく標準的な計算を使用すると，その計算には何年もかかるでしょう。したがって，並列計算は本質的に重要です。

私たちは，最大尤度に基づくプログラムを開発し，それをIntel社のDeltaで実行し，500種類の有機体を含む微生物に関する系統図を作成しました。それは，この技術を用いて，15ないし20種類以上の有機体に関する系統図を作成した初めての試みです。これは，最終的に並列処理を必要とする限定されたアプリケーションの１つです。また，共通にみられる現象として，負荷を均等化するという重大な問題に突き当たりました。元々のアプリケーションは，PCN（Cサブルーチンを起動）の方言の１つで再構築されました。私の考えでは，これは，コミッティッド・チョイス論理プログラミングを共通かつ重要な意味で使用することを反映していると思います。本質的には，これはコーディネーション言語と呼ばれている言語です。この意味で重要な言語に関しては，それを既存のアプリケーションコードに正しくリンクできなければなりません。CとFORTRANの両言語，および多数のコミュニティに対するこれまでの投資はあまりにも巨額であり，無視することはできません。

最後に，私が現在取り組んでいるもう１つのアプリケーション分野，すなわち地理的情報システムについて簡単に申し上げます。製造業の企業を対象として，店舗の設置場所に関するコンサルタント業務を行っています。これは，データベースの作業です。具体的には，地理情報データベースを作成し，そのデータベースを使用して統計的モデルをサポートします。驚いたことには，この作業を進めるにあたって費用効果が最も高いのは，Prologを使用する方法でした。現在では，50万以上の節が入っているデータベースを日常的に作成しています。そして，作成したデータベースは，比較的安価なハードウェアで非常にうまく動作します。確かに，システムに占めるハードウェアのコストは人件費よりもはるかに小さく，しかもプロジェクトのコストの大半を占めるのは明らかに人件費です。また，柔軟性が持つ利点は，非常に重大です。つまり，アプリケーションが正しく実行されれば，少なくとも何億ドルもの資金を節約できます。店舗の設置場所を正確かつ迅速に決定することができれば，人件費やワークステーションの運用コストは，ほとんど取るに足らないものになります。重要なことは，正しい答をできるだけ早く出すことです。とはいえ，このような作業にPrologを使用することには，明白な欠点があります。しかし，現時点では，この方法が費用効果が最も高いのは興味深いことです。

これで，私の最初の発言を終わらせていただきます。

Kowalski：どうも有り難うございました。続きまして，古川康一さんにお願いしたいと思います。

古川：FGCSプロダクト株式会社の科学部門担当の販売課長の視点からお話したいと思います。

当社の製品を販売するには，アプリケーションが必要です。そのアプリケーションは，他のどのようなアプローチを採用しても，当社の製品には太刀打ちできないというようなものでなければなりません。

このようなアプリケーションの品質にとって最も重要な要求とは，当社のマシンの最大の特徴である記号計算に大きく依存する点です。次に，「大量の記号計算は，現実の生活のアプリケーションにおいて重大なのか」という疑問がわいてきます。「はい」というのが，それに対する私の答です。現在のところ，大量の記号計算が必要となるアプリケーションは小数です。しかし，大量の記号計算が障害となって，これまで誰も解決しようとしなかった隠れたアプリケーション分野がたくさん存在します。そのような問題の中には，データベースからの知識の獲得，データ分析，人間のゲノム分析などの多くの組合せ的な問題，そして診断，設計，仮設生成，制約充足問題など，いわゆる逆問題が含まれます。論理の用語では，逆問題は非演繹的な問題として解釈されます。そのような問題では，マシンの構造から入出力動作を推測するのではなく，入出力動作からマシンの構造を推測しなければなりません。そのため，このような問題を逆問題と呼びます。

論理プログラミングには重要な側面が２つあります。１つは知識を表現するツールという側面であり，もう１つは推論エンジンとしての側面です。知識表現の側面では，論理プログラミングは，開放的な世界を「失敗としての否定」で表現できます。一般的には，論理プログラミングの価値は演繹的な能力だけにあると信じられていますが，それは真実ではありません。安定モデルの理論は，「失敗としての否定」を含めて，一般論理プログラムに関するモデル理論を提供します。また，この枠組では，仮設生成，帰納法，および類推を自然に形式化できます。

推論機関の側面からは，自動探索，およびトップダウン探索戦略とボトムアップ探索戦略の組合せは，強力な推論能力を提供します。

「失敗としての否定」と仮設生成を扱う問題に関しては，井上さんが演繹的な手続きによる計算方法を提案しました。すなわち，「失敗としての否定」と仮設生成の問題は１階論理の証明問題に変換されます。したがって，非演繹的な推論問題を演繹的な問題に変換できます。その結果，この方法によって，仮設生成に関する逆問題のクラスを解決するための直接的な方法が提供されます。

私たちのプロジェクトにおいて，並行論理プログラミングの最も重要な特徴は，もちろん並行性の表現力です。この特徴は，汎用的な並行プログラミングにとって非常に重要です。PIMのオペレーティングシステムを含めて，数多くの有益で複雑なアプリケーションがKL1言語で書かれています。私たちの並行論理プログラミング言語の明らかな弱点を克服するために，あらゆる解を計算するための探索プログラミング方法をいくつか考案することによって，完全性またはそれと等価的な意味でOR並列性を回復しようとしました。その結果，並行論理言語の完全性の特性を実質的に回復することができました。したがって，私たちの並行論理言語は，リアクティブネスと完全性の両方を備えています。

論理的な逆問題，またはそれと等価な仮設生成問題をPIMで解決するには，適切なプログラミング法が必要です。仮設生成推論は演繹的推論へ変換可能なので，KL1言語で効率の良い並列定理証明器を実現するだけで目的を達成できます。幸いにも，ボトムアップな手続きで定理を証明する，モデル生成定理証明器(MGTP: Model Generation Theorem Prover)と呼ばれる非常に効率的な並列定理証明器をすでに開発しています。

これまで指摘した長所から，並行論理プログラミングは並行性と探索の両方において十分な表現力を持つ，とごく自然に結論することができると思います。また，並行論理プログラミングは並列的な計算と高い親和性を持つため，人工知能システムだけでなく，21世紀のより汎用的な情報システムにとってもふさわしい基礎を提供します。並行論理プログラミングと並列処理に基づく技術は，人工知能に関する将来の数多くの問題を解決する上で大きな可能性を秘めています。

Kowalski：ありがとうございました。非常に多くの興味深い考えを手短にまとめた，すばらしいプレゼンテーションだったと思います。では，次のパネリストの内田俊一さんのプレゼンテーションをお願いします。

内田：皆さん，こんにちは。このプロジェクトを推進してきたメンバーの１人として，ここにお集まりの多くの皆さんがこのプロジェクトに示してくださった大きな関心に感謝を捧げるとともに，皆さんがこのプロジェクトを非常に高く評価してくださったことをとても誇りに思っています。

私は，他のパネリストの方々とは多少異なる見解を紹介したいと思います。私の専門はハードウェアアーキテクチャです。したがって，私たちが構築してきたマシンによって，工学にとどまらず定理証明器などの科学的なアプリケー

21世紀へ向けての知識情報処理(KIPS)

1．KL1言語とAYA言語による並列記号処理

論理プログラミング＋オブジェクト指向プログラミング

私の驚き --> KIPアプリケーションには十分な並列性がある。
1,000PE --> 10,000PEs --> あるいはそれ以上

一層の努力

- 並列アルゴリズム，負荷分散法など
- KL1, PIMOSを他の多くのMIMDマシンへ移植
- 言語レベルまたはOSレベルで既存のソフトウェアとインタフェースを持つ

2．１階論理に基づくKRLでの知識プログラミング

証明器 --> より「汎用的な論理」プログラミングのための実用的な推論エンジン

「汎用的な論理」プログラミング言語
--> 人文科学的なAPまたは認知科学的なAPでの知識表現のための「アセンブラ」

何かもっと必要なのだろうか？　それは，状況理論か様相論理か？

図9

ションを含めて，新しいアプリケーション分野への道が開かれたことは大きな喜びです。ここでは，このスライド（図９）が示す２つのテーマについて述べるとともに，論理が，将来の計算機科学にとって非常に適切かつ有益であることを強調したいと思います。

まず最初に，21世紀の知識情報処理について述べたいと思います。そして次に，２つのテーマについて研究を進めるべきだということを主張します。その１つは，もちろん並列記号処理です。私たちの場合には，KL1言語とそれを拡張した高水準言語のＡＹＡを使用します。ＡＹＡは，論理プログラミング，つまりKL1言語とオブジェクト指向プログラミングを組み合わせたものだと考えてください。

ごく最近のことですが，定理証明器やタンパク質の構造分析システムなどのいくつかのアプリケーションプログラムにおいて，多数のプロセッサで非常に高い利用率を達成することができました。この結果には驚きました。正直に申し上げて，この結果が出るまでは，多くのプロセッサを使用することによって，ほとんど線形的な速度の向上を達成できるとは考えていませんでした。

今では，ハードウェア研究者がソフトウェア研究者に対して大規模なマルチプロセッサ，たとえば１万個を上回るPEを提供できれば，再び線形的な速度の向上を達成できることに自信を深めています。したがって，並列アルゴリズム，負荷分散法，および数多くの並列処理手法の研究や開発を精力的に行うべきだと思います。精力的な研究開発によって，間違いなく巨大な市場が開拓されると思います。近い将来では，市場開拓のスピードは，今後数年のうちに登場する多くの商用MIMDマシンにKL1言語とPIMOSを導入するなどの努力に大きく影響されると思います。

もう１つ重要なのは，言語レベルとオペレーティングシステムレベルで，私たちのシステムと既存のソフトウェアをつなぐためのツールまたはソフトウェアを提供することです。このような努力によって，新しい市場の開拓が加速されるだけでなく，並列処理アプリケーションに関するさまざまな研究開発の分野が拡大します。そのような技術は，今後５年ないし10年以内に実現される可能性があります。

では，その後に何が起こるのか予測してみたいと思います。最近，私は，技術をかなり巨視的な視点からとらえるようになりました。これは，管理的な立場になったせいだと思いますが，人によっては，単に私が年をとったせいだと言っています。いずれにしても，最近は，このような新しい技術が実現されたらどうなるのか知りたいと考えています。そして，将来の知識処理がどんなものなのか想像しています。

このスライド（図９）の下半分には，将来のイメージが示されています。このイメージによって，論理プログラミングは，特にハードウェアアーキテクチャにとっては素晴らしい土台であると自信をもっています。もちろん，他の意見，たとえばオブジェクト指向モデルが現実のアプリケーションにモデル化には欠かせないとするPeter Wegnerさんの意見などにも賛成です。しかし，ボトムアップの視点，つまりハードウェアの視点からは，どちらがハードウェアレベルで実現されるモデルの核となる演算または計算なのか決定するのは困難だと思います。

１階論理または何らかの限定された論理モデルの方が，私たちにとっては明白かつ理解しやすいものです。したがって，やはり同じ土台，すなわち論理の方を採用します。コンピュータ設計者の今後10年間の夢は，汎用的な論理プログラミングのための，より汎用的な推論エンジンに取り組むことだと思います。現在のところ，非常に特定された，または限定された論理プログラミング言語しか存在しません。

汎用的な論理プログラミング言語が日常的なツールとして手に入ったならば，より直接的に

アプリケーションに集中できます。まずマシンとオペレーティングシステムに注目して，それを徹底的に理解しなければなりません。するとその後で，アプリケーションを効率的にプログラムできるように，アプリケーションに目を向けることができます。

汎用的な論理プログラミング言語が実現した場合には，前述の最初の手順は省略して，知識プログラミングを取り込むために各アプリケーションに目を向けるだけでよくなります。知識プログラミングが主な課題の場合には，1階論理言語は，もはや高水準言語ではなくなってしまうと思います。その言語は，知識プログラミングのためのアセンブラであると言う人が出てくるかもしれません。

最近，私たち，特に私自身は，このような夢を抱き続けているのです。私のプレゼンテーションを短縮したいと思いますので，スライドの多くを省略させていただきます。

将来は，当然，非常に強力な並列記号処理システムを実現できると思います。また，このスライド（図10）に示すように，そのマシン上で高レベルの推論エンジンを実現できると思います。現在，いくつかのエンジンが，定理証明器またはオブジェクト指向データベース言語のための言語インタプリタとして開発の途上にあります。しかし，論理を統一化された土台として維持している限り，自然科学のみならず人文科学や認知科学の分野においても，知識情報処理を押し進めていくことができると思います。

アプリケーション開発者が，たとえば法律的な推論システムなどについて，多種多様な形式でさまざまな知識を記述するのを観察すると，

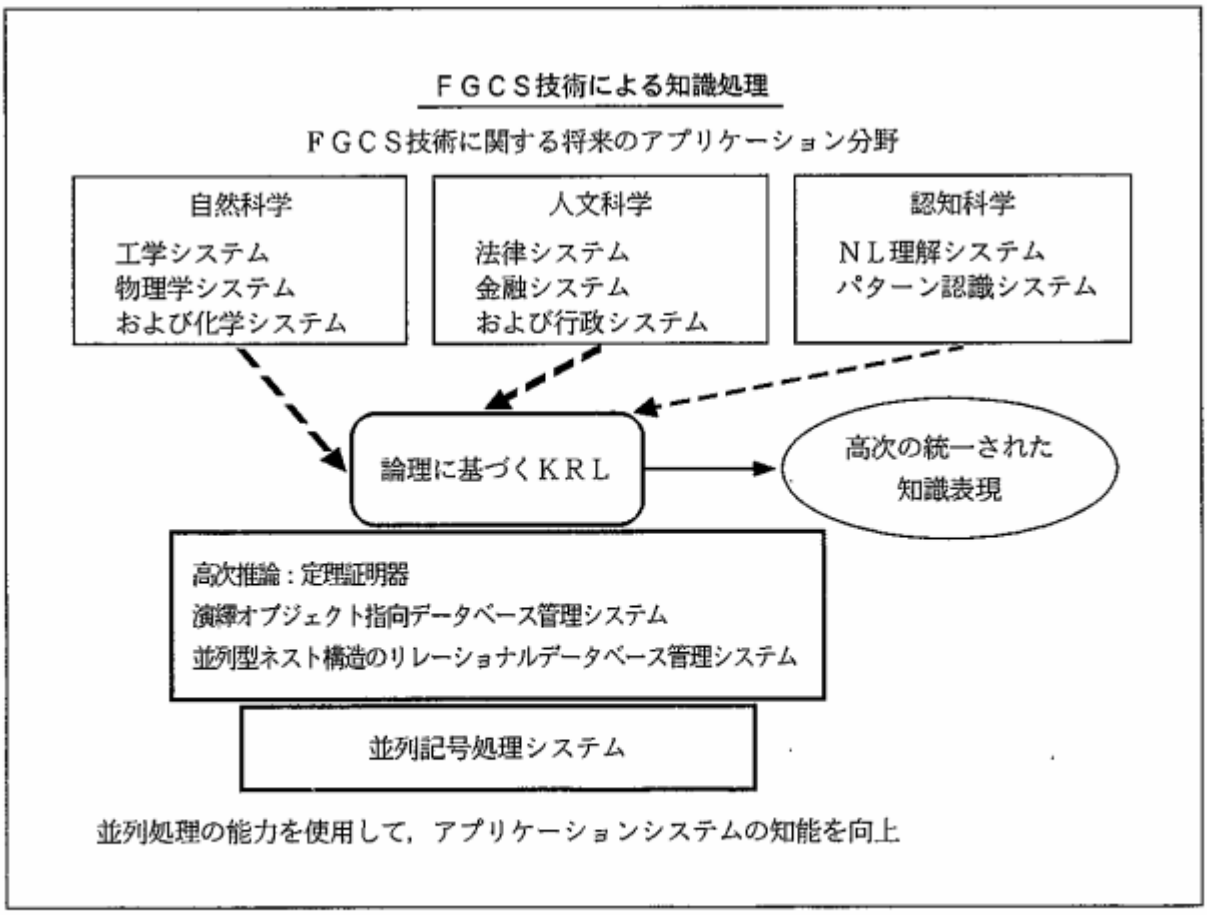

図10

さまざまな科学的な分野から取り出され，表現された知識の相違点と品質を比較することができます。現在は，知識の品質や特徴を表すのに「知識」という１つの言葉だけを使用しています。各種の分野の知識を区別することはできません。したがって，この統一化された言語を使用して，各分野の知識を比較できるような客観的な見方が必要なのです。

私たちは，知識処理の新しい時代へ進むことができると思います。今でもさまざまな意見がありますが，論理は特に知識情報処理に関する研究で中心的な土台として使用できると考えています。どうもありがとうございました。

Kowalski：ありがとうございました。ここで，司会者の立場を離れ，１パネリストとしてお話しさせていただきます。私の同僚，特にICOTの研究仲間にならって，明確かつ適切にお話ししようと思います。

私のプレゼンテーションでは，FGCSの技術として論理プログラミングを選択したことから発生した問題のいくつかに焦点をあてます。また，FGCS技術の将来の見通しは楽観的である，と考える理由についてもいくつか述べます。

プログラミングのために論理プログラミングを使用することと，プログラム仕様のために論理プログラミングを使用することの間に生じるギャップを説明するために，まず並び替え問題に目を向けてみます。次に，これまでの進歩の重要性に注意を向けるため，および論理プログラミングの貢献を指摘するために，非単調推論のテーマを取り上げます。

最後に，論理プログラミングと特定の様式の自然言語との比較を行います。特に，ロンドンの地下鉄の緊急時の注意書きなど，公共の注意書きに使用されている言語に注目します。これは，人間が実行するために自然言語で書かれたプログラムと見なすことができます。そのような公共の注意書きは，さまざまな人々が全く同じように理解するようにできている点で，プログラムに似ています。このため公共の注意書きは，きわめて明確かつ簡潔な自然言語様式で書く必要があります。そのような自然言語の様式と一部のコンピュータ言語の様式の間には，非常に大きな類似性があると思います。最大の類似性は論理プログラミングと比較した場合に見られますが，命令型言語やオブジェクト指向言語と比較した場合にもある程度の類似性が見られます。

このような例から結論を導き出すと，論理プログラミングは，計算に関しては，本質的な限界を持っているかもしれません。しかし，その限界は人間のさまざまな行動を規制するために，自然言語を明確かつ簡潔に使用する場合の限界と同じものだといえます。しかも，自然言語を明確に，正確に，そして効果的に使用するのがきわめて困難なのと同様に，良い論理プログラムを書くのも困難なことがあります。

最初の例を見てみましょう。（図11）

仕様

宣言型
Y is a sorted version of X
if Y is a permutaion of X
and Y is ordered
もしもYがXの順列で
Yが順序通りならば，
Yは順序通りに並べ替えられたXのバージョンです

手続き型
to sort X into Y,
generate a permutaion Y of X
test that Y is ordered
Xを順序通りに並べ替えてYにするには
Xの順列Yを生成し
Yが順序通りに並んでいるかテストしてください

図11

皆さんの多くは，宣言型で明示的に書かれた，この並び替え問題の仕様をこれまでに何回もご覧になっていると思います。論理プログラムとして解釈すると，この仕様は手続きになります。実際には，使用モードによっていくつかの異なる手続きとなります。入力データオブジェクトXを与えられると，この手続きはドントノウ非決定論的手続きとなります。そして，Xの順列を生成し，その順列の順序が正しいかテストします。この手続きはかなり非効率的で，nを入力Xの長さとすると，nの階乗の複雑さを持ちます。仕様を実行できる可能性があるという理由で最初にPrologなどの論理プログラミング言語に魅せられた多くの熱狂的な人も，このような非効率性がわかると幻滅を感じます。

しかし，クイックソートなどの効率的なアルゴリズムを使用すれば，非効率性を解消することができます。クイックソートは，以下のように論理プログラミング形式で書くことができます。(図12)

このアルゴリズムは，宣言型と手続き型の両方で記述できることに注意してください。通常，論理プログラムの記述で使用する従来の宣言型の形式の方が，手続き型よりも受け入れられやすいことについては，本質的な理由は何もありません。また，論理プログラミング言語が，宣言型と手続き型の両方を許してはならない理由も何もありません。

このように，プログラムまたは仕様を表現するために論理プログラミングを使用することは，

プログラム

手続き型　to sort X into Y
　　split X into X1 and X2,
　　sort X1 into Y1,
　　sort X2 into Y2,
　　merge Y1 and Y2 into Y
　　Xを並べ替えてYにするには
　　XをX1とX2に分割し，
　　X1を並べ替えてY1とし，
　　X2を並べ替えてY2とし，
　　Y1とY2を併合してYにしてください

宣言型　Y is sorted version of X
　　if X can be split into X1 and X2
　　and if Y1 is a sorted version of X1
　　and Y2 is a sorted version of X2
　　and Y is a merge of Y1 and Y2
　　もしもXをX1とX2に分割することができ，
　　かつY1はX1を並べ替えたバージョンであり，
　　かつY2はX2を並べ替えたバージョンであり，
　　かつYはY1とY2を併合したものであるならば，
　　YはXを並べ替えたバージョンです

図12

知識を宣言的または手続き的に表現することとは異なります。論理プログラミングの支持者は，初心者も専門家も，必ずしもこの違いを認識しているわけではありません。これが，論理プログラミングが，これまであまり受け入れられなかった１つの理由かもしれません。多くの論理プログラミングの支持者は，プログラムと仕様の違いを区別できなかったし，また論理プログラミングを高レベルかつ高い効率で使用する方法を学ぶことができませんでした。

このように，ソフトウェア工学上の問題があります。しかし，それは克服できる問題です。また，この問題を源として，将来は１つの論理プログラミングパラダイムに基づく非常に豊富なプログラミング様式を達成できると思います。

では，２番目のテーマである非単調デフォルト推論に移ります。過去10年間において，人工知能と論理プログラミングの両分野では，論理によってデフォルト推論を表現できるようになったことで，論理を使用できる可能性が大きく向上したと思います。従来の論理では一般的に適用できる精密で正確な文だけしか表現できないため，形式論理を経済や政治などの分野で現実に発生する問題に適用した場合には，現在のところはまだ問題に直面することになります。

非単調論理では，デフォルト時に適用できる「all birds fly（すべての鳥は飛ぶ）」というような不正確な文で，「ostriches do not fly（だちょうは飛ばない）」というような例外によって上書きされるような文であっても，推論を行うことができます。この種の推論は，「失敗としての否定」を使用して，論理プログラミング形式で以下のように効果的に表現できます。

X can fly if X is a bird
and not X is unable to fly
X is unable to fly if X is an ostrich
Xが鳥で，かつXが飛べないと言えないならば，Xは飛ぶことができます
Xがだちょうの場合には，Xは飛ぶことはできません

論理プログラミングによって，デフォルト推論に関する便利な形式化，適切な意味論，および効果的な証明手続きが得られます。これによって，コンピュータを使用するにしろ，またコンピュータの助けを借りずに直接人間が行うにしろ，論理を実用的に適用する私たちの能力は飛躍的に向上します。この能力の向上の重要性こそ，論理プログラミングが将来計算の内外を問わず重要な役割を果たす，もう１つの理由だと思います。

しかし非単調推論のテーマには，さまざまな研究団体が類似性に気づかずに同じような問題と取り組むという，研究の社会学的な側面に関するもう１つの教訓があります。いわゆる「イェール射撃の問題」は人工知能の分野で発生したよく知られた例であり，デフォルト推論への多くの主要なアプローチに対する反例として証明されています。この問題を扱った文献は多数あるものの，論理プログラミングでの「失敗としての否定」がこの問題に対する自然で効果的な解を提供することがわかるまでに何年もかかりました。その解の本質は，状況計算(situation calculus)おけるフレーム公理の否定条件を解釈することです。

property P holds in the state after action A, if property P holds in the state before action A, and not action A terminates P
もし特性Pがアクションの前の状態で成立し，かつアクションAが「失敗としての否定」の意味でPを終了させないならば，特性Pはアクションの後の状態で成立します。

イェール射撃の問題の経緯は，異なる研究団

体相互の競争と協力という社会学的な問題が，技術的な利点と同じくらいの重要性があることを教えてくれます。一方，FGCSプロジェクトが養ってきた国際的な協力体制も，同じ文化圏でさまざまな技術研究を代表する研究者が集まる場合に比べて，文化圏が異なっても同じ技術研究の世界から研究者が集まって協力する方が，お互いの理解が深まる可能性があることを教えてくれました。

最後の例は，ロンドンの地下鉄の緊急事態に関する注意書きです。これは，人間が実行するプログラムと見なすことができます。将来のコンピュータ言語は現在よりも自然言語により近いものになる可能性があるので，この例は，将来のコンピュータ言語の特徴の一部を示してくれると思います。(図13)

最初の文は手続き型ですが，論理プログラミング形式では次のような宣言型の解釈であることに注意してください。

You alert the driver if you press the alarm signal button.

非常ボタンを押すと，運転手に警告します。

この例は，「論理プログラムは宣言型と手続き型の両方の構文を許すべきである」という，先ほどの私の提言を裏付けてくれるものです。他の例は，論理プログラムでは，論理プログラミングの意味を損なうことなくオブジェクト指向の構文を有益な形で許すことができることを示

ロンドンの地下鉄の注意書き

緊急時の処置

Press the alarm signal button
to alert the driver.
運転手に警告するには，
非常ボタンを押してください。

The driver will stop immediately
if any part of the train is in a station.
車両の一部がすでに駅構内に入っている場合には，
運転手は即座に運転を停止します。

If not, the train will continue to the next station,
where help can more easily be given.
そうでない場合には，駅に入った方が支援を求めやすいので，
地下鉄を次の駅まで運転します。

There is a £50 penalty
for improper use.
いたずらで非常ベルを押した場合には
50ポンドの罰金が課せられます。

図13

しています。

また，他の計算パラダイムも，他の公共の注意書きに類似性をもっているように思われます。たとえば次の注意書きを見てください。

Please give up your seat if an elderly or handicapped person needs it.

年配者または身体に障害のある人に席を譲ってください。

この注意書きは，条件－アクション・プロダクション規則の形式です。また，論理プログラミングの用語では，統合性制約として解釈できます。同様に，次の注意書きをみてください。

Do not obstruct the doors.

ドアの前に立ち止って通行の邪魔をしないでください。

この注意書きは命令型文の形式です。しかし，それは統合性制約としても解釈できます。

このように公共に注意書きで使用される言語，すなわち人間が実行するプログラミング言語として使用される自然言語は，多くのさまざまなコンピュータパラダイムと類似性があります。しかし，それは各種のコンピュータパラダイムを見事に統合して1つの統一言語にします。論理プログラミングを自然な方法で拡張することで，宣言型の構文だけでなく手続き型の構文を包含し，また手続きだけでなく統合性制約を包含して，コンピュータで実行可能な形式の自然言語に形式的な基礎を与えることができると思います。

最後になりましたが，適切に拡張された論理プログラミングと公共の注意書きで使用される言語の類似性から，人間の意志の伝達において厳密な自然言語に見られる限界と同じように，論理プログラミングには計算パラダイムとしての限界があります。

自然言語を明確かつ正確に使用するのがきわめて困難なのと同じように，効果的な論理プログラムを書くのは困難だと思います。人間が言葉や言葉以外のものを通してお互いに意志の疎通をはかるのと同じように，将来のコンピュータ言語も，たとえば絵を指したりまたは描いたりするなど，言葉による言語的な通信手段と言語以外の計算メカニズムを統合化する必要があると思います。とはいえ，人間の意志の伝達に使用される自然言語と同様に，論理プログラミングが計算の将来において重要な役割を果たすことは確実だと思います。

これで私のパネル，およびパネリスト全員のプレゼンテーションを終了します。今から，聴衆の皆さんにも討論に参加していただきたいと思います。パネリストが漏らした点を指摘してくださっても結構ですし，また私たちの挑発的な発言に対する意見を述べてくださっても結構です。

Randy Goebel（アルバータ大学，アメリカ）：挑発的な意見の1つ，特にWegner教授の発表に対して意見を述べたいと思います。

教授が述べられた歴史には誰かが抜けていると思うのですが，その人物をどのように評価するのかお尋ねしたいと思います。1930年代に，「世界は，オブジェクトとそれらのオブジェクト相互の関係として捉えることができる」と考えた人がいました。その視点で世界を見るとしたら，世界が具象的であろうと抽象的であろうと，論理に関連する仕様，つまり論理の構文と証明理論を使用することができます。私たちが注目しているモデル相互の関係としての私たちの理解は，真実という点で討論することができ，また証明理論を用いて推論をシミュレートすることができます。この理論を提唱した人はAlfred Tarskiです。オブジェクト指向に発案

したのは，彼だと思います。Wegner教授がモデルと呼ぶものと，頭の中で抽象的にモデル化する世界との関係として論理プログラマが見るものとの間には，根本的に混同があると思います。

Wegner：形式的なモデルと記述型のモデル間には違いがあります。オブジェクト指向プログラミングではモデル化を非形式的に記述し，また表現やシミュレーションを意味するのに，数学的なモデルではなく項モデルを使用します。記述型のモデル化は，モデル理論的なモデル化とは異なっています。Tarskiは，形式的に正当化できるモデルを研究しましたが，その結果は記述型のモデルには適用できません。現実世界のモデル化における記述の柔軟性を実現するためには，私たちが行っていることの形式的な正当化をあきらめなければなりません。私の話の中で，形式的なモデル化と非形式的なモデル化の間のトレードオフについて具体的に触れ，形式的なモデル化では，コミュニケーションの柔軟性（リアクティブネス）を犠牲にする必要があることを示しました。

Kowalski：他のパネリストの中で，この質問に対して意見を述べたい方はいらっしゃいますか。

Gallaire：推論とモデル化は独立しているわけではないので，質問者が述べられたそれらの違いを私自身理解しているかどうか確かではありません。たとえば制約プログラミングを取り上げた場合，現在のその使用方法は，モデル化でも推論でもなく，その両方だと思います。制約メカニズムを通して得られる推論能力は，新しいモデル化能力を与えてくれます。制約解決器はあるかないかによって，問題を同じようにはモデル化しません。したがって，その解決器は，同時に大きなモデル化能力を備えていることになります。ですから，モデル化と推論が完全に直交しているとは言えないと思います。この２つの関係は，はるかに複雑だと思います。

オペレーションズリサーチには，多くの例が見つけられます。オペレーションズリサーチで古典的な問題をモデル化した方法，および制約条件を使用する場合に問題をモデル化する方法を見てください。それは全く異なっています。もっと効率的である場合も，また効率が落ちる場合もありますが，いずれにしても全く異なっています。

Wegner：この点については永久に議論が尽きないので，別の点に話題を移した方がよいかと思います。

Kowalski：では，次の質問者の方，お願いします。

Bob Keller(Harvey Mudd College, U.S.A.)：２人のパネリストの方に質問したいと思いますが，その前に，聴衆の皆さんにとって，また多くの人にとって異説だと思われる意見を述べたいと思います。

私たちは，論理プログラミングと並列コミッティッド・チョイス・プログラミングという２つのパラダイムを混同し続けている，という大きな間違いを犯していると思います。もし，これが正式な会議であれば，この２つの概念を区別するための用語法をただちに採用すべきだという提案を行っているであろうと思います。なぜならば，それは一般の人にとって非常に紛らわしいものであり，また第五世代コンピュータプロジェクトに関するいくつかの混乱の原因でもあると思うからです。確かに，このプロジェクト自体は，この２つのパラダイムの分野においてすばらしい業績を挙げました。これらは，ある意味でお互いに他を解釈できることは認めますが，やはりこの２つの概念を区別することが重要だと思います。

私の質問は，Gallaire博士とOverbeek博士に対するものです。どちらの方もプレゼンテーションの中で，パラダイムについて触れられた

と思います。それは,Gallaire博士の場合には論理プログラミングを意味し，一方Overbeek博士の場合にはコミッティッド・チョイス・プログラミングを意味していたと思います。両博士は，それぞれのパラダイムは，他の数多くのプログラムまたはさまざまなパラダイムで書かれたプログラムに対するマスター，コーディネータ，ないし指揮者としての役割を果たし得ると述べられました。その意見の根拠は何なのでしょうか。この2つのパラダイムが，他の対象候補のパラダイムよりも適切だと考えられた理由は何なのでしょうか。

Overbeek：私の個人的な意見では，並列処理はいくつかの個別の領域に分割できます。1つの領域は，大きな数値また記号アプリケーションの領域ですが，たいていは数値に関する領域です。現時点では，並列処理の大半の作業はこの領域に集中しています。この状況で大量の並列アプリケーションを見た場合に，コミッティッド・チョイスはそのような問題を解決するためのメッセージパッシング・アプローチを管理する優れた方法だと思います。この場合には，時間の大半は，かなり低水準の言語で書かれた，かなり限定されたルーチンの集合に費やされます。

さて，並列処理には，全く異なる世界があります。この世界では，並列処理は，小数のプロセスを持つワークステーションによって特徴付けられます。研究者は，この世界での並列性の追求に時間を取られたくないと思っています。この場合には，私の意見では,OR並列Prologが理想的な解を与えてくれると思います。

しかしどちらも全く異なる市場であり，経済的に見て断然重要なのは数値アプリケーションを扱う大量並列処理だと思います。PCNなどで見られるものは，その市場に対するコミッティッド・チョイス技術のアプリケーションであり，それは重要な市場だと思います。

Gallaire：私が考えている種類のアプリケーションないし私が言及するところの計算は，正確に言えば，現在ワークフロー言語を使用しているものです。新しい製品やパッケージが市場に登場しており，これを使用すると，ネットワーク上に分散される既存のアプリケーションでユーザ固有のアプリケーションを記述することができます。新製品によって何ができるかを見てみると，複雑なアプリケーションの構築のためにやりたいことからは，まだかけ離れています。同期化の手法は存在しません。必要なのは，コミッティッド・チョイスの機能と制約を組み合わせたものだと強く信じています。1ユーザとして申し上げれば，どの順序で事が行われるのかはっきりとはわかりません。私はシステムを解決したいと考えていますが，アプリケーションこそシステムの解決です。これが，プログラミングについて私が言及した点です。

Furukawa：先ほどのプレゼンテーションの中で，一部ですが，あなたのご質問にお答えしたことになるとおもいます。私は,Prologよりコミッティッド・チョイス，つまりドントノウ非決定性の方が好きです。それには2つの理由があります。1つは，先ほど述べたように，コミッティッド・チョイス言語のプログラミング方法論では，探索パラダイムまたは完全性を効果的に回復できるからです。それが第1の理由です。もう1つの理由は，言語のアーキテクチャ，つまりハードウェアを含む情報システムを階層構造として設計したい場合には，並列型コミッティッド・チョイス言語はむしろ低水準言語のレベルに位置することです。後は，先ほど私が述べたこととおそらく同じです。上位レベルの記述は，コンパイルした後，下位レベルの記述に変換されます。そのような階層構造のアーキテクチャは重要で，問題解決のために何らかの酵素を開発するようなものです。その酵素を入れると，上位レベルの言語で記述された

プログラムは消化されて，並列ハードウェアによって処理されることになります。

Richard Peer（Weizmann Institute,イスラエル）：並行論理プログラミング言語の分野の研究者の間で意見が一致する数少ない点の1つは，その分野を表す並行論理プログラミングという名前です。したがって，まず，この分野の研究者は，10年間におよぶ研究の後に得られた，この最低限のコンセンサスを尊重するようにしていただきたいと思います。

次の点ですが，Peter Wegner教授は非常に優れた発表をされたと思います。教授は自らを部外者と位置付けられました。言ってみれば，この分野に直接関わっていなかった研究団体が最終的に理解できたものの1つは，リアクティブネスを表現する必要がある場合には不確定性がなければならない，という基本的な原則だと思います。この意味においては，並行論理言語は，A'UM,アクター，Adaを始めとして，並行性を表現しようとする言語と異なっていません。Adaをコミッティッド・チョイスADA,A'UMをコミッティッド・チョイスA'UMと，そしてSmalltalkをコミッティッド・チョイスSmalltalkと呼ぶ人は誰もいません。並行性を表現するこれらの言語には，すべて不確定性が必要です。

最後の点ですが，現在進行中の研究はまだ開始されたばかりですが，この2つのパラダイムを結合しようという関心や意気込みが高まっています。それは，Pandoraとそれに類似した言語の方向を目指す研究です。これらの言語は，Prolog言語の長所と並行論理プログラミング言語の長所を統合するための，理論的な見地および実用的な見地から技術的な解が存在することを示しています。したがって，2つのパラダイムはそれほどかけ離れていないのかもしれません。

Keith Clark（Imperial College,イギリス）：できれば，いくつかの点について述べたいと思います。まず，Peter Wegner教授の発言について述べたいと思います。教授はコミッティッド・チョイス言語など，モデル化を行うことができる言語に対して「完全性」と「完全性の欠如」という表現を使われました。この表現は，問題の本質をそらすものだと思います。関数プログラミングを行う場合には，コミッティッド・チョイス言語にはプログラミングのサブセットがあります。補完的なガードを用いたときが，それに当たります。

さて，関数型言語が宣言的であることには賛成してくださると思います。また，関数型言語の実行戦略と評価戦略は完全なものになるでしょう。したがって，その場合には完全性が得られます。

実際に並行論理言語は，論理変数，すなわち不完全メッセージの概念を持っているのでより一層強力です。それは，オブジェクトのモデル化に使用されるものです。補完的なガードによる決定論的なケースを緩和します。Steve Gregoryはこのガードの補完性のことを,「ガードの十分性」と呼びました。これは，少なくとも常に1つの答が得られるという，すばらしい論理的な特性を持っています。1つ心配なのは，探索，または1つのプロセスにおいて完全性が欠如している場合には，節に対する条件付きコミットの評価を行い1つの解を見出すと，他のプロセスは解をみつけられないという点です。しかし，それでもその特性を保持したまま決定的プログラムの緩和をすることができるのです。

興味深いことに，緩和された集合において，あらゆるオブジェクト指向プログラミングをエミュレートできます。その場合には，そのような完全性が良い完全性として得られます。

以上が，その点に関する批評です。

HerveさんとRossさんが取り上げられたと

思う点を，さらに追求してみたいと思います。実際のところ，ある意味でこの新しい技術を市場で販売しなければならないという意見に，今では全く賛成しています。日本における第五世代コンピュータプロジェクトは，今後も続けていかなければならないと思います。そのプロジェクトで並行記号アプリケーションに関して開発された，非常に有力な方法論について説明しなければなりません。その方法論は，公開される必要があります。現在では，その方法論が，「粒度の細かい複合型アプリケーションを並列マシンで開発するための優れた技術」である理由を世界に示さなければなりません。

Uchidaさんが，そのような計画について述べられたのを聞いて非常にうれしく思いました。その計画では，UNIXが動作するありふれたハードウェアにKL1とPIMOSを移植し，UNIXまたはCを介して他のアプリケーションとのインタフェースを確立します。その計画はきわめて重要だと思います。実際に，Rossさんが言われたように，主なアプリケーション領域は，コーディネーション言語としての並行論理プログラミングになると思います。それ以上に，分散人工知能アプリケーションのための言語として，これらの言語が分散型で実現されることを願っています。全く新しい技術，つまり協調型活動支援機能(CSCW)の研究開発が始まりつつあります。その分野においても，課題がたくさんあると思います。それについては，この計画ではまだ言及されていません。以上です。

Kowalski：この並行論理プログラミングに関する分野は，確かにFGCSプロジェクトで重要な前進が見られた分野です。しかし，マスコミの注目を集めている人工知能や知識情報処理，また将来のコンピュータ市場の重要な先駆けとなりうる並列推論マシンなど，他のFGCS技術についても議論をしたいと思います。

Pierre Deransart(INRIA，Rocquencourt，フランス）：計算機科学の歴史は非常に浅く，論理プログラミングに至ってはさらに歴史は短くなります。歴史が非常に浅く，またある意味で有望な可能性を秘めている論理プログラミングの将来について，そのような質問をするのは無理があると思います。しかし逆に，20年近くに及ぶ開発研究の後という意味では，賢人に対して「将来はどうなるのか教えてください。私たちの研究は十分だったのでしょうか」などという質問を発するのは当然でもあります。では，私が研究の視点からごく手短かにお答えしたいと思います。私たちはその答だけでなく，自分たちがしていること，および自分たちにできないことは何かを知っていると思います。

私たちは，基本的には信頼性に取り組んでいることを知っています。そして，ソフトウェア技術の信頼性を向上させたいと思っており，また新しい問題も解決したいと思っています。たとえば，制約条件論理プログラミングによって，従来のソフトウェア技術では取り扱うことが困難な，これまでにはない問題を解決できるようにしたいのです。したがって，何が必要なのかわかっていますし，その方向に向かって進歩しようと努力していますが，その限界も知っています。しかし，ソフトウェア業界は，作成できないものを作成しようと試みているいう意味で特殊な業界であり，私たちは困難な時代に生きているのも事実です。建造物を建設しようとする場合は，最新の技術を駆使して，ビルや橋梁が倒壊しないことを保証できます。しかし，ソフトウェア技術では，私たちは不可能だとわかっているもの，つまり安全なソフトウェアまたは絶対安全なソフトウェアを作成するように求められています。だからこそ，その方向に向かって進むことは興味深いのです。

もう一方で，この業界はこれまで以上に研究が必要であるという意味で，興味深い局面に来ています。私たちは困難を抱えています。何を

すべきなのでしょうか。私たちがすべきことを判断して，または探すのを手伝っていただけるでしょうか。先ほど申し上げたように，ある意味で私たちに何ができるのか部分的にわかっていると同時に，何ができないかもわかっています。しかし，限界はわかっており，それが大きな限界であることも理解しているが故に，その研究を続けるためには資金が必要です。物理学者に今年の末までに核融合を発見するように依頼する場合，それまでに核融合を発見できなくても，研究者は資金を手に入れるということを考えてみてください。これははばかげたことですが，起こりうることです。起こらないはずがありません。しかし，計算機科学では，より一層の発達を遂げるには資金が必要であるという点を無視するのが，当然であるように思われています。

次の質問は，私たちは1歩前進したのか，10～15年前に比べてソフトウェアの信頼性という点でかなりの進歩があったのかという点だと思います。この質問に対しては，直接答えることはできません。ソフトウェア業界の状況を見てみると，パリの論理プログラミング国際会議における産業展示会などさまざまな出来事がありました。また，ソフトウェア業界の視点から見ても，多くの興味深いアプリケーションと満足できるアプリケーションが紹介されました。最近では，ロンドンで開催されたPrologアプリケーションに関する展示会および会議があります。そこで発表されたアプリケーションの大半は産業用アプリケーションであり，それは論理を大幅に拡張したものではなく，古典的なPrologに基づくものでした。将来，論理の大幅な拡張が絶対欠かせないことはわかっています。一方，現在のアプリケーションは，古典的なPrologに基づいているという事実も承知しています。古典的なPrologの使用は標準化への道であり，業界にとってよいことです。

実際の影響力を知りたい場合には，例が1つあります。Prolog 1000を見てください。Imperial CollegeのChris Mossは，あらゆる産業アプリケーションを再現しようとしています。彼に，アプリケーション例を送ってみてください。そうすれば，Prolog 1000に基づく多くのアプリケーションができるでしょう。これは非常によいことです。それは将来にとって興味深いことだと思います。以上です。

Kowalski：ありがとうございました。パネリストの中で，どなたか答えたい方はいらっしゃいますか。

Overbeek：信頼性についての意見，すなわち私たちが求めているものが信頼性だとおっしゃいましたね。それは非常に視野の狭い見方であると思います。確かに信頼性は追求するに値するものですが，それが第五世代コンピュータプロジェクトのビジョンではなかったと思います。計算は，解，すなわち科学的な疑問に対する答えを見出すためのツールであると思います。万物の祖先の特徴は何だったのでしょうか。生命を得たオートマタの構成要素は本来何だったのでしょうか。地球の温暖化は現実に起きているのでしょうか。このような疑問を解決するために，マシンが存在するのです。信頼性などの概念は，本質的な考え方を追求するための従属的な概念なのです。計算機科学者は，信頼性などの概念を重視する傾向があります。私は，科学者として，それを超えて，もっと一般的でスケールの大きいビジョンを持たなければならないと思います。

Murry Shanahan（Imperial College，イギリス）：この点は，実際に信頼性の問題に関係しています。Peter Wegner教授が指摘された形式的な妥当性検証の点に話を戻したいと思います。もしも教授が搭乗されている航空機の設計技師が物理学の原理を理解していないことを知ったとしたら，教授は帰国時に飛行機の中

で安心した気持ちでいられるでしょうか。論理は，工学の分野での物理学に似ています。計算機科学において論理プログラミングをパラダイムとして使用するポイントは，論理は論理プログラミングに非常に近いということです。一方，オブジェクトなどを使用すると，論理との距離は大きくなります。これが，その距離を縮めるためには，論理プログラミングのようなパラダイムに注意を向けなければならないと考える理由です。計算機科学で飛躍的に複雑なシステムを構築しているときに，形式的妥当性検証の概念を拒否することは多少不心得なことだと思います。

Wegner：わざと不心得なことをしたのです。それは，純粋に論理的な証明は，現実世界の実体を考慮していないため，間違っていることがよくあると思うからです。飛行機では，墜落しないという形式的な証明があるという理由だけでは安心できません。偶発的なできごとを常識によって非形式的にテストする方が，形式的な証明よりも普通はずっと重要です。安全性のためには，物理学の法則に従うことは重要ですがそれで十分ではありません。なぜなら，考慮しなければならない事柄のうち，証明に馴染まない実際的な事柄が数多く存在するからです。プログラムの正しさに関する初期の論理的な証明には，後で誤りだとわかったものが多数ありました。したがって，私は，「証明されたプログラム」によって制御される飛行機，つまり「その正しさの証明が誤りであることが後からわかるようなプログラム」によって制御される飛行機の墜落の犠牲にはなりたくありません。

溝口文雄（東京理科大学，日本）：日本人は，私のように物静かな人が多いようですが，あえて意見を言います。

私は，アプリケーショントラックのパネルの議長を務めましたので，その御報告をしなければなりません。そのパネルでは，知識表現，制約論理プログラミング，帰納的論理プログラミング，および並列プログラミングから始まる，いわゆる、パラダイムに関して議論しました。すべてのパラダイムを併合するのは難しかったのですが，プログラミングの基礎としてCを使用するという点である程度意見が一致しました。Catherine Lassezさんは，制約論理プログラミング，略してCLP，そして最後に将来のCまたはC++について，優れた図解を紹介してくださいました。言語が本来提供するパラダイムとは異なるパラダイムが必要な場合は，そのような種類のパラダイムを開発するために何をすべきなのでしょうか。パラダイムが１つのものであり，計算がCのような従来の言語だとしたら，パラダイムと計算をどうやって区別するのでしょうか。

並列言語と論理プログラミング言語の役割について質問があります。それを思考パラダイムとして，またはパラダイムの開発研究の言語デバイスとして使用してもいいのでしょうか。すべての計算の基礎がCやC++によって提供される場合，21世紀には何が起こるのでしょうか。

Kowalski：パネリストの中でどなたか答えていただけますか。

古川：言語の重要性は，自身の考えをプログラム化するのを手助けすることにあります。Prolog, KL1, GHCなどの高水準言語は，思考を助けてくれます。CまたはC++の長所は，その効率性にあるかもしれません。しかし，良いコンパイル手法はたくさんあると思いますし，そのような長所は高水準言語のスキーマに取り入れることができると思います。重要なのは，ソフトウェア作成の生産性であり，思考の容易さであると思います。したがって，高水準言語の役割は非常に重要です。

内田：OR並列PrologとAND並列Prologの違い，および論理プログラミング方法論とCやFORTRANでの従来のプログラミング方法論

との違いについて意見を述べたいと思います。

FORTRANまたはCを使用しているときは，ユーザはハードウェアに近い位置にいます。まず，使用対象のハードウェア資源を詳細に考えなければなりません。しかし，論理プログラミングを使用している場合には，ユーザの位置はFORTRANやCよりも多少高くなります。その場合，メモリの最大容量などのハードウェア資源の細部は無視できます。このため，アプリケーション問題を直接取り扱うことができます。

しかし，アプリケーションのモデル化方法を考えなければなりません。これは，プログラミングする上で，アプリケーション問題のモデル化という別の段階です。アルゴリズムの設計，フローチャートとプログラムの作成，ハードウェア資源の細部の考慮などを行わなければなりません。つまり，多くのことを同時に考えなければなりません。

論理プログラミングでは，この手続きはかなり単純です。それは高水準言語の長所の１つです。OR並列論理プログラミングは，資源管理という点で，AND並列論理プログラミングよりもかなり高レベルです。Bobが紹介してくれた最後のスライドは，論理プログラミングの１つの限界を示唆しています。彼は，自然言語や絵を直接扱うために，１つの単純な論理プログラミング言語を使用しようとしています。そのアプリケーションの範囲は，たった１つの言語でカバーするには広すぎます。それは，おかしなアプローチだと思います。

現在の論理プログラミング言語はまだ低水準言語であり，今でも主にハードウェアシステムの制御に使用されていると思います。まだ，効率的なOR並列論理言語は提供されていません。言語がハードウェアシステムの細部をどのように隠すのかについて考えなければなりません。この点は，OR並列言語とコミッティッド・チョイスAND並列言語と比較するときに非常に重要です。それは最重要事項だと思います。

Kowalski：私の方から，ごく手短にお答えすべきだと思います。私がお見せした最後のスライドは最悪の場合のシナリオです。まだ少し時間がありますので，あと１人の発言を受け付けます。

Catherine Lassez（IBM，T.J.Wattson，U.S.A.）：私は，滝口さんの発言内容をちょっと明確にしたいだけです。私のスライドでは，Cは，実際にはプログラミング言語Cではなく制約条件を意味しています。Cを使用する場合もありますが，他の領域に適用するには論理プログラミングの線に沿って形式モデルを使用することを強くお勧めします。

Kowalski：ありがとうございました。このパネルディスカッションで，まだ取り上げられていない分野が１つあると思います。それは，並列推論マシンの研究分野です。その重要性について意見を述べたい方はいらっしゃいますか。

Wegner：その点については，Bobがその重要性を感じているようなので，発言者としてBobを指名します。

Kowalski：ありがとうございます。ICOTの並列推論マシンに関する研究は，これまで認識されているよりも本当にはるかに重要だと思います。

私の主張の要点は，並列推論マシンは，人工知能アプリケーションという狭い範囲に関して，難解で従来のものとは異なる論理プログラミング言語をサポートするために特別に設計されたものではないという点です。並列推論マシンは，むしろ主流の計算モデルをサポートする汎用並列マシンです。それは，並行論理プログラミング，すなわちPIMによって直接サポートされる計算モデルは，論理プログラミングの形式であるのと同様に，オブジェクト指向プログラミングのモデルであり，またCSPやCCSなどの並行性に対する主要アプローチのモデルでもあ

るからです。

この視点から，FGCSのPIMに関する研究は，規模，革新性，および成功度の点で，最高レベルの並列コンピュータアーキテクチャとその関連ソフトウェアの開発であったということができます。したがって，それが将来の並列コンピュータアーキテクチャの開発に重要な影響を与える点については，少しも疑っていません。

Wegner：記述されたオペレーティングシステムは，たとえばプロセス移行など，多くの主流の機能を備えています。FGCSは，論理プログラミングに特に依存していない多くの問題の内，他のパラダイムに容易に移行できる問題について検討してきました。これは非常に重要です。

Kowalski：それでは，このパネルディスカッションを終える時がやってまいりました。閉会にあたって，このディスカッションを代表するいくつかの結論を要約したいと思います。

まず最初に，論理プログラミングとオブジェクト指向プログラミングを融和させなければならないという点で意見が一致しました。別の言い方をすれば，それは論理プログラミングのdon't careとdon't knowとのギャップを埋めなければならないということになると思います。

第2は，将来のFGCS技術の可能性を評価するには，技術的な長所だけが重要なわけではないことに気づくべきだということです。社会学的な考慮も重要です。特に，Ross Overbeekさんが主張したように，プログラミング言語Cは，技術的な理由だけで現在の計算における支配的な地位を築いた訳ではありません。

第3に，今週，私たちは，ICOTが紹介かつデモンストレーションで説明した数々のすばらしい成果を見てきました。このような短い期間に，その成果を完全に評価することはできません。たとえば非決定性のdon't care形式とdon't know形式とのギャップを埋めるためのいくつかの提案によって，このディスカッションで討論してきた問題の多くがすでに解決されている可能性さえあります。このような成果を評価するには，もっと時間が必要です。

最後に，この会議の参加者全員を代表して，非常にすばらしい10年間を与えてくださっただけでなく，この会議を準備して数々のすばらしい研究成果を発表してくださったこと，そしてコンピュータの将来を紹介してくださったことに対してICOTに感謝したいと思います。ありがとうございました。