コンカレント・システムの理論と実際

1988年12月2日（金）
15：30〜18：00

座長：(Shapiro)コンカレント・システムの理論と実際をテーマとするパネル討論を始めます。私はイスラエルWeizmann科学研究所のEhud Shapiroです。今回のカンファレンスは，大変水準が高く刺激的でありましたが，このパネル討論もそれにふさわしいものとし，このカンファレンスの有終の美を飾りたいと願っています。

コンカレント・システムというテーマに関し，広範囲な視点と背景を代表する著名な研究者に集まっていただきました。パネリストの皆さんの意見と討論，また，会場からの質問に対する解答から，多くを学べるものと期待しています。

パネルの進行方法は，まず，各パネリストが10分ずつ発表し，それぞれについて短い質疑応答を行います。全員の発表が終わった後，討論に入ります。討論は他のパネリスト，もしくはフロアからの質問に答える形で進め，パネリスト全員の意見を順々に聞いてゆきます。会場で質問や意見のある方は，遠慮なく手を上げて下さい。適当なところで，私が討論を区切りますので。

William Dally博士はカリフォルニア工科大学で博士号を取得し，いろいろな業績をあげていますが，中でも，Torusルーティング・チップの設計者として知られています。現在はMIT（マサチューセッツ工科大学）でJマシンという粒度の小さいコンカレント・コンピュータを構築するための研究グループの責任者をつとめています。Carl Hewitt博士は，MITで博士号を取得し，現在はMITの教授です。博士の最大の業績である並列計算向きの，「プランナー」および「アクター」の形式化については，後ほど話が聞けるでしょう。

Robin Milner博士は，エジンバラ大学の教授で，ロンドン王立学会特別会員です。博士には，プログラミング言語ML,MLの標準バージョン（現在研究が進行中），さらに，通信システム計算法CCSに基づく並列性理論の研究等の業績があります。

上田和紀博士は，東京大学で博士号を取得し，現在はICOTの研究員です。博士は並列論理型言語GHC(Guarded Horn Clauses)の設計者として知られていますが，GHCは，ICOTをはじめとする世界中のいろいろな研究所で行われている研究の基礎となっている言語です。

David Warren博士はブリストル大学の教授で，Prologの逐次型・並列型インプリメンテーション技術に関し，重要かつ根本的な成果をあげています。

カリフォルニア工科大学のGeoffrey Fox教授が，アプリケーション側を代表して出席する予定でしたが，残念ながら，欠席です。当然のことですが，このパネル討論は，皆さんが得意とし，関心を抱いている分野，つまり，コンカレント言語，並列モデル，並列型アーキテクチャという話題を中心に進めます。

DALLY：まず，現在，どのようにして並列システムの構築が行われているかをお話します。

主な取り組み方として，2種類の方法があります。

第一の方法は，単純に最新型のマイクロプロセッサを持ってきて，それらの間をネットワー

クでつなぐというやり方です。これはあまりうまくいきません。それにはいくつか理由がありますが，その最大のものは，それらのマイクロプロセッサが，過去40年以上にわたって逐次計算のためにチューンアップされてきた逐次コンピュータの進化の結果であるという点です。それらは逐次型プログラムにおけるLIFO式の実行のためのスタックを備えています。また，I/Oアーキテクチャは，何十ミリ秒もの待ち時間があり，転送ブロックが何千バイトにもなるようなディスク等のデバイスに合わせたものであります。このため，I/Oチャネルを通じて通信ネットワークに結びつけようとしても，メモリ・インタフェースで処理しようとしても，どちらともうまく行きません。通信メカニズムのニーズと合わないからです。

第二の方法は，好みの計算モデルを採用し，もっとも当然と思われる方法で，それをハードウェアで実装するというやり方です。最適化とエンジニアリングは，まあまあうまく行くとしましょう。それでも，こちらの方法にも問題があります。１種類の計算モデルしか実装していないからです。２～３ヵ月後には，それを変更したくなるかもしれませんし，誰かが別の計算モデルで作成されたプログラムを持ってくるかもしれません。

これら２種の方法では，ハードウェアは何か，ソフトウェアは何か，という点から出発し，即座にインプリメントしてしまっています。つまり，ハードウェアとソフトウェアの中間に位置する基本メカニズムが何であるかを見極めずに，インプリメントしてしまうのです。

Carlが後で話すアクター・モデル，アーキテクチャのセッションでいくつかの話のあったデータフローモデル，共有変数を介して通信し合う論理型プログラミング言語も含むと考えられる共有メモリアーキテクチャ，そして，Waltz博士の今朝の講演で実例が紹介されたデータ並列型プログラミング，こういったさまざまな計算モデルを見てみると，その土台となるハードウェア面で，どれもが３つの必要事項を備えています。

まず，通信が必要です。計算モデルが分散していてもいなくても，マシンは物理的に分散していますから，情報をマシン中のある箇所から別の箇所へ移動させなければなりません。同期を取ることも必要です。いつ計算結果を生成したいか，必要な入力はいつ得られるのか，こういった条件にもとづき，タスクを実行する時点を決定します。さらに，共有メモリ中のセル，データフロー・プログラム中の同期点，あるいは，アクター・プログラム中のオブジェクトなどに名前をつけるため，何らかの命名法が必要です。

アーキテクチャ構築においては，正しいインタフェースを設定するという難関があります。これはアーキテクトの仕事です。コンピュータの設計者は，そのインタフェースをインプリメントします。インタフェースは，通信の同期メカニズムと命名法で決まりますが，これはどちらも，これまでに発表された計算モデルのほとんどを効率良くインプリメントし，容易にサポートするものでなければなりません。我々の研究の中間結果からは，ある特定の計算モデル向けに設計したマシンの1／2から1／3程度の性能で，これが可能であることが分かっています。

現在，Intel社との共同研究でMITが構築しようとしているJマシン上で実装されているメカニズムを，２～３紹介しましょう。まず，高速メッセージ処理です。これは，私がカリフォルニア工科大学で開発した高速ネットワークを，さらに超えたものです。今では，何千ものノードから成るマシンの端から端まで，メッセージが数ミリ秒で届くようなネットワークの構築が可能です。それは問題の一部にすぎません。メカニズムという観点から見ると，メッセージ伝

達とは，メッセージの送信処理や，それがノードに到達した時のメモリ割り当て処理，メッセージ到達時のタスクの同期処理を含みます。そのため，今挙げた4つに対して何かしなくてはなりません。これまでにさまざまな送信メカニズムを試し，現在の送信命令を選択しました。これは，実装コストと通信効率の兼ね合いを考えた場合，良い妥協点であると判断したためです。これを使い，6語から成るメッセージをマシンの端から端まで伝達するのに，3命令の実行と同じコスト，200ナノ秒しかかかりません。次に，メッセージを宛先に届けるためのネットワークですが，負荷分散用ハードウェアや，ボトルネック回避のためにルーティングを変更するような賢いネットワークを構築するためのいろいろなアイディアを検討しました。その結果，最も重要な点は，ネットワーク中のオーバーヘッドの軽減，つまり，ネットワーク自身の高速化であることがわかりました。そこで，ネットワークの機能を減らし，エンジニアニングに重点を置きました。そして，最終的に，メッセージがノードに入ってから出てくるまで，わずか2～3ゲート分の遅延にまで短縮することができました。

メッセージが宛先に到着すると，次は，それをどこかへ置かなければなりません。これがメモリ割り当ての問題です。逐次型アーキテクチャでは，実行環境中のレコードやスタックにメモリ割り当てを行うように特化されています。並列アーキテクチャでは，メッセージキューの中の並列環境に対してのメモリ割り当て機能やその他それに類する機能が必要です。

メッセージが到着すると，そのメッセージの到着と，それが到着したノードで進行中の計算との同期を取るということが問題になります。メッセージキューをここに示した箱と考えると片方の端からネットワークのメッセージを取り出すと考えることができます。しかし，プロセッサ側の観点で見れば，これは実行待機リストです。どのメッセージも，実行待機リスト中のタスクに対応するものと考えることができます。タスクは待機リストの先頭から取り出される必要があります。適当な機構を用いれば，このタスク選択は1クロック・サイクルで済み，メモリからのフェッチも1回です。タスク選択命令ポインタの指定による制御環境と，セグメント・レジスタの切り替えによるアドレス環境が生成されます。この機構は皆さんがお考えになるほど高価なものではありませんが，それがなくとも，この会議でWarren教授が発表したスヌーピー・キャッシュ・プロトコルのような，他の物でも構築できます。

タスク選択により起動されたハンドラは，例えばスヌーピー・キャッシュ中の1行の状態を更新するようなものや，そのキャッシュプロトコルのために別のメッセージを送出するというような簡単なものから，より大きなアドレス環境を設けることにより，完全なプロセスにまで成長するものもあります。Jマシンのオペレーティング・システムでは，あるメッセージを完全なプロセスに成長させるのに，1マイクロ秒もかかりません。

これらの計算モデルをサポートするために必要な他の同期メカニズムとしては，データの存在に関する同期が考えられます。これはたとえば，データがまだ存在していないことを示すタグをつけることで達成できます。この場合，そのデータを参照しようとすると，タスクは中断されます。

時間が10分しかありませんので，ここではお話しませんが，最後のメカニズムはネーミングです。データフロー・マシンにおける待機・照合メモリや，Warren教授が言及した特殊キャッシュ群など，特殊アーキテクチャ中の多くのものが，いわゆる名前変換機能の特殊な例と言えます。それらでやりたい事は，何らかのキーと

関連づけてデータを取り出すことです。これらの計算モデルすべてに対し，特定の位置や特定のデータ要素のネーミングに必要な条件を考え，それから，それらモデルをインプリメントするための最も根元的なメカニズムを組み立てるとしましょう。

我々が選択したのは，キーをデータに変換するための一般的変換メカニズムと，メモリ中のデータを保護する何らかのメカニズムというセットです。我々は，セグメンテーションを選択しました。

これらメカニズムを選択する基準としては，やたら大げさなメカニズムや，実現しようとする計算モデルやアルゴリズムを直接表現するメカニズムを狙っているわけではありません。特定のアルゴリズムを専門とするネットワークを採用することも，非常に容易です。これは，ネットワークでそのアルゴリズムの論理相互接続が可能なためです。しかし，通常は，汎用のネットワークを選択し，アルゴリズムの論理的なネットワークをそれにマッピングする方が，効率が良いのです。この場合，論理的な通信ネットワーク中の異なる論理リンク間で(物理的な)通信リソースを共有できる場合もあるため，かえって性能が良くなることがしばしばあります。

これらのメカニズム選択における最大の難しさは，小さいオーバーヘッドで実装可能なメカニズムを選ぶということです。

Jマシンの通信メカニズムではメッセージの組立てと送信，4000ノードから成るマシンの端から端までの伝達，最終的に6語から成るメッセージの目的地でのバッファリングの合計に，約2マイクロ秒を要します。これは，コネクションマシン/モデル2より2桁，インテルIPSC-1ハイパーキューブより3桁良い性能です。

Jマシン同期メカニズムでは，基本的ハンドラの起動には1クロック・サイクル，タスクの生成，中断，破壊，再開には，1マクロ秒弱しかかかりません。「効率」の良し悪しの判断の1つに，このオーバーヘッドと，このメカニズムで計算モデルをインプリメントする際の変換オーバーヘッドがあります。

「効率」のもうひとつ判断材料は，ハードウェアリソースがいかに有効に利用されるかということです。我々のアーキテクチャのコスト/パフォーマンスはどれぐらいになるのか。並列マシンにとっての「効率」の真の基準はこの点であり，文献で頻繁に使われるプロセッサの利用度ではありません。N個のプロセッサを持つマシンを構築するとして，ある1個のプロセッサ上での稼働にT-1という時間がかかり，N個のプロセッサでTという時間がかかるとすると，マシンのコスト/パフォーマンスは，1台の面積に対する1台の時間とN台の面積に対するN台の時間との割合です。速度向上度を，単にプロセッサ数で割った割合ではありません。

最後に，並列マシン構築においては，この効率は，しばしば1よりも大きくなる，という最後の公式について考えてみたいと思います。こうなる理由は，パワフルなプロセッサの構築はそれほど高価なものではないからです。Jマシンのインプリメンテーションに使っている技術では，プロセッサはチップの面積の約10の1しか占めていません。これはかなりパワフルな10MIPSの36ビットCPUです。チップはほとんどがメモリ・チップです。実際，粒度が小さい並列コンピュータとは，プロセッサあたりのメモリ量が非常に小さいマシンであり，必ずしも，非力なプロセッサや，単一ビット・プロセッサ，あるいはその類を備えたを持つマシンを意味しません。ですから，これらのメモリ・チップを，90年代のジェリービーンと考えるなら，期待できることは，多数の計算モデルを効率良くインプリメントできるような，正しい汎用メ

カニズムを定義することにより，これらジェリービーン要素からジェリービーン・マシンを構築できるだろうということです。多くの計算モデルはこういったメカニズムに対応させることができるので，移植性のある並列プログラムの作成は可能です。もはや，１台のマシンの上だけでしか走らないようなプログラムを作成する必要はありません。同じメカニズムをサポートするさまざまなマシン上にプログラムを移すことができるのです。ご静聴ありがとうございました。

座長：ありがとうございました。２～３分ありますが，パネリストの皆さん，会場の皆さんから，何か意見や質問がありましたら…。

WARREN：簡単な質問をひとつ。各プロセッサのサイズに対するメモリ量の割合について，どう思いますか。そういったマシンでは，プロセッサが必要とするメモリ量に，最適値があるのでしょうか。または，それをどのように決めると良いのでしょうか。

DALLY：これはたいへん良い質問で，これまでもたびたび聞かれました。Gene Amdahlが1960年代に提唱した経験にもとづく法則がいくつかありまして，彼は従来の逐次型プロセッサ・アーキテクチャを研究していたわけですが，プロセッサにとって好ましい状態を保つには，１MIPSの性能につき最低１メガバイトのメモリが必要だと言うのです。先程のジェリービーン・コンポーネントでは20MIPSのプロセッサ中に，メモリはわずか64キロバイトで，これはAmdahlの法則にはあてはまりません。この違いがどこから来るかというと，Amdahlは，必要とするメモリ量がI/Oシステムの性能によって決まるマシンを想定していたわけです。つまり，充分なメモリというのは，遅いディスクからメモリのページをスワップする間，プロセッサからフル稼働できる量なのです。しかし，メモリ量に関する真の指標は，プロセッサがメモリを必要とする時，ある一定の遅延時間内に，どれだけのメモリを使用できるか，という点にあります。ですから，効率の良い通信メカニズムをインプリメントし，論理仮想アドレス空門をサポートするためのネーミング機構を用いれば，たとえメモリが物理的には分散していようとも，非常に小さいローカル・メモリだけで，プロセッサをフル稼働させることが可能です。我々は10^4から10^5バイトの間で実験を行っています。多くのアルゴリズムに対し，これで充分なようです。そして，真に重要な意味を持つ値は，プロセッサあたりのメモリ量ではなく，マシン全体のメモリ量であると思われます。ある時点でマシン中に，問題解決に必要なメモリ量を持っていればI/Oが問題になるような事態は避けられるということです。

（会場から）：質問があります。あなたのマシンでは，ローカル・アクセス時間とグローバル・アクセス時間の割合はどのくらいですか。Jマシンでのグローバル・アクセスの速さはどの程度ですか。

DALLY：我々のマシンということですと，特定のインプリメンテーションについて話すことになります。それは根本的な限界ではありません。我々のマシンでは，ローカル・アクセスに100ナノ秒かかります。グローバル・アクセスですと，あるメッセージを往復させるとして，宛先での読出し書込みメッセージ・ハンドラが，メモリの読出しか書込みを行い，メッセージを戻すまでに，平均２マイクロ秒かかります。つまり，割合は20：１になります。

座長：ありがとうございました。後ほど討論の中で，さらに質問を受け付けます。

HEWITT：今の話に私が話したかったことがうまく説明されていました。それは，逐次型計算と並列計算の違いです。並列計算では多くのことが同時に起きます。逐次型計算では，別の場所にいる誰かを見張らなければならないと

いう問題はありません。ここで私は，アクター・アーキテクチャに関する研究の背後にある原理について，お話したいと思います。

目的は2つです。

最初の目的は，並列が物理法則によってのみ制限されるような並列システムをインプリメントする超並列性です。これは，たった今Billy Dallyが話したようなことです。つまり，我々は，超並列性という概念に，次第に近づいてきているということです。並列性の限界はどこにあるのでょうか？並列性の限界となるのは，物理法則であり，また，皆の手帳の大きさでしょう。しかし，中には性能曲線の外郭にまで手を延ばしたいという人もいるので，基本的な限界に関心があります。つまり，限界は物理法則です。そして，第二の目的は，強靭さです。強靭さとは何でしょうか。ここで，私は強靭さは信頼性とは異なると言って，皆さんを困らせようと思います。

信頼性とは，完全に繰返し可能であることを意味します。システムが常に同じことを同じ方法で行い，ある程度の許容範囲内で，同じ結果が得られます。強靭さはそれよりも一般的で，つまり，信頼性しか使わないような範囲を越えることをしようとする，ということです。

超並列性は厳しい必要条件を要求します。その性能は物理法則によってのみ制限されるべきであり，それには，Billy Dallyが話したような優れた工学が必要です。さらに，超並列性は，維持しなければなりません。システムが高速に動くようにするだけでは不充分です。並列性は計算の進行につれて，超動的に維持されなければなりません。超並列性には，並列型計算の形態が，アプリケーションの形態に適応する，すなわち，動的に適応する必要があります。

我々は，超並列型計算に必要とされる能力を持つ万能の並列基本要素として，アクターを開発しました。

さて，ここで重要な考え方は，アクターが実は，それぞれの実装法に関する数学的モデルだ，ということです。アクターは特定のプログラミング言語ではありません。たとえば，私の研究室の学生の一人は，並列マシン上で完全なGHC言語のアクターに基づく実装にかかりました。したがって，プログラミング言語からの独立は，それに適応力をもたらし，ある種の言語上の衝突を避け，また，言語上の衝突に取り組む基礎を提供する，非常に重要な考え方です。我々は，アクターを使って並列システムの性質についての仕様を記述し推論します。コンピュータ・サンエンスの実用では，数学的な仕様を与える代りに，数学的たとえば，企業は監査組織を内部に持ち，企業全体があるべき姿をなしているかをチェックしています。アクターのアプローチは，先の並列システムをチェックできる別の並列システムを提供するというものなのです。なぜなら，現実世界で仕様が成り立つようにするには，メカニズムを生成するしか無いからです。したがって，アクター・システムは実装だけでなく，仕様記述にも使われます。

Bill Dallyはこれらのものとのインタフェースをとる必要性について少し話しました。特殊目的プロセッサと汎用プロセッサの両方を備えたマルチコンピュータの構築について話していました。ここでしなければならないことは通信インタフェースを指定することです。ソフトウェア・システムは，上位レベルからインタフェースレベルまで構築し，その上に位置します。ハードウェア・システムは，ゲート，メモリ，プロセッサ，アービタ，等々を使い，下位レベルからインタフェースレベルまで構築します。介在する通信インタフェースはそのような内部結合が可能になるように指定されなければなりません。アクターは，まさにこの種のインタフェースを提供します。このようなインタフェースは，アクター・システムにさまざまな

利点を提供します。そのひとつは，この種のシステムがマルチプロセッサからマルチコンピュータへ，ローカルエリア・ネットワークからワイドエリア・ネットワークへ移る際に，得られるマシン透過性です。このインタフェースにより，システム中にさまざまな異機種の特殊プロセッサを備えることができるようになります。新しいシステムの接続が可能になります。計算は，それがどのくらいの規模になるか，前もって決定することはできません。この新しいオープン・システムの原理は，新しいコンピュータそして最新のコンピュータをそれに接続できるということです。この通信インタフェースは，物理的性質と通信能力から切り離すことにより，超並列性を促進するのです。これにより，後に工学的仕様に変換される一種の数学的仕様記述メカニズムを提供します。

並列計算は逐次計算よりもはるかに面白く，また，それとは全く異なります。最も重要な相違点のひとつは，並列計算が非決定性を有するとういう点です。非決定性とは結果がどうなるか何も「確定されない」ことを意味します。アクターの非同期的動作は，必然的に不確定性を生みます。前もってどれほどの情報を収集しようと，結果がどうなるかはあいまいです。シカゴの口座に80ドルあったとして，東京でその金を引き出すため，電子式資金移動を行なうとしています。ロンドンでは別の誰かが50ドル引き出そうとし，ボストンでも誰かが50ドル引き出そうとしています。この人たちのうちの誰かはがっかりすることになる，とは言えます。さらに，構造と初期の構成についてどれだけ詳しく知っていても，この電子式資金移動システムの動作は非決定的です。

衝突によってマクロ分析のくい違いが生じます。世界中で並列に動いているさまざまな人が，相入れない目的を持ち，噛み合わない行動様式を取ります。多くの場合，彼らは，衝突が起きることを前もって認識していません。そこで，関係者は協議して，銀行に残った最後の50ドルは引き出さないと確認する，ということになるかもしれません。しかし，この協力的組織があっても，最後の50ドルを引き出すはめになります。他の関係者に知らせる前に，資金を調達しなければならない緊急事態が起こるからです。

つまり，このようなことから，最終的に強靭さを定義することになったのです。すなわち，非決定性と衝突に直面しても，「堅牢性」は責任を放棄しません。ここが信頼性と異なる点です。強靭であるためには，システムはより豊かな資源を備える必要があります。最初の方法がうまくいかなくても，その状況を判断し，何をすべきかを決断しなければなりません。したがって，強靭さこそ，我々が分析したいと思うことです。非決定性と衝突は常に存在します。しかし，私はまだ責任が何かを説明していません。責任が何かを定義し，それらがどのような関係になっているかを説明することは，並列システムの理論の基本的作業のひとつです。

まず私が話したいのは，うまく行かない場合を発見することです。これは，今回集まった面々の中では，論議を呼ぶことは必至です。皆さんの中には，反撃のためにメモを取っている人もいるようですね。

具体的には，論理型演繹には，２つの根本的問題があると主張したいのです。

最初の根本的問題は，論理型演繹は強すぎるということです。いかなる「現実の」状況の公理化にも，矛盾が起こります。これらの矛盾はさまざまな衝突し合う責任の結果として生まれます。

論理型演繹法は，強すぎるだけでなく，弱すぎる面もあります。共有口座から複数の関係者が金を引き出す例に戻って考えてみましょう。関係者の一人は，要求している金を引き出すことができません。この結論には演繹的に到達す

ることができます。しかし，誰かが金を獲得するかについては，演繹的に判断できません。そのステップについて，演繹的に結論を出すことはできません。そして，我々のメッセージ交信システムでは，逐次型モデルの全順序事象列と異なり，並列計算は，互いに順序が交差しあう半順序事象列を持ちます。この絡まり合った半順序事象列により，並列システムの実装には，論理型型演繹法は使えません。そして，このような理由で，私は並列計算は“演繹に制御を追加したもの”とは等しくない，と言いたいのです。演繹にどのような制御を加えようと，必要とする決定の演繹はできません。

次のステップは，コンピュータ・サイエンスという分野にとって何を意味するか，という点です。コンピュータ・サイエンスは，新しい研究分野であり，まだ基礎固めの最中にあります。ひとつのアプローチは，私がアルゴリズミクスと呼んでいる方法ですが，計算およびコンピュータの基礎を，数学的論理に置くというものです。このアルゴリズム的アプローチは，動作環境の性質も含め，実行すべきタスクの正確な仕様を記述し，正しさを証明すると共にタスクを実行するためのコードを作成し，その手続が仕様と一致するという証明を行います。しかし，正確な仕様記述は，数学的問題領域に対してのみ可能です。オープン・システムに対する厳密な仕様記述を行なうとすると，衝突し合う決断があるため，常に矛盾が起こります。したがって，コンピュータ・サイエンスの基礎を，純粋にアルゴリズムに置くという方法は，うまく行きません。

GHCに関して言えば，良い点はそれが論理型プログラミングではないということです。もしもGHCが論理的プログラミングであったなら，それはアルゴリズミクスの中に閉じ込められてしまっていたでしょう。マルゴリズムは数学にのみ有効で，オープン・システムには役に立ちません。ABCL,ÁUM,PARLOG等と同様GHCも論理型プログラミングではありません。論理型プログラミングであれば，これらの仕事に必要な能力がありませんから，そうでないということは良いことです。

最後に，コンピュータ・サイエンスの新しい基礎として必要なものを指摘したいと思います。そろそろ時間切れですので，今日はこれについてあまりお話できません。しかし，システムが現実世界で動けるようにするために，コンピュータ・サイエンスの新理論中に必要となる概念に触れたいと思います。我々は強度試験のようなものを知る必要があります。どの関係者が決断しようとしているか，あるいは，責任を持っているかを，知る必要があります。権限について，また，権限を他の権限がどう支配するかについて，知る必要があります。衝突と協力のようなものについて知り，また，システム中のある関係者が他の関係者に対し，何かを表現しなければならない場合の表現という概念を持つ必要があります。以前，AIから表現の心理学的概念が提出されました。それは，知的動作者の精神構造と，外界でのでき事の状態との間のやり取りです。そのような人工知能における定義は，協力的かつ競争的な社会組織には適用できません。そこで，社会科学および組織的理論に基づいて，以前の心理学的概念を補うものとして，新しい定義と新しい科学を獲得しなければなりません。このようにして，我々は大規模並列計算に必要な基礎を創造することができるでしょう。

座長：ありがとうございました。Carlは2時間分の議論の種を撒いてくれましたが，今はパネリストの皆さんからコメントをいただくだけにして，発表を進めることにしましょう。どなたか？…無いようですので，Robin Milner教授の発表へ進みます。

MILNER：私が話そうとしていたことを，

だいぶCarlが話してしまったようです。私は社会学はやめにして，代数の話を付け加えましょう。

まず，我々が探し求めているのは，記述的科学だということです。そして，逐次計算のように，計算の一部のみをカバーするのではないものを探しているということです。それは計算の全領域，さらに，それをはるかに超えたところをカバーします。さて，この記述的科学を3つの面から眺めてみたいと思いますが，おそらくその理想像は，決して見つからないでしょう。第一は普遍性，第二は基本要素，第三は構造です。

最初にわかったのは，たしかCarl Petriが最初に発見したことだと思いますが，記述に3つのレベルを設けてはならないということです。フォンノイマン型のモデルでは，何らかの形で，我々に記述のレベル分けを強制しました。というのは，マシンの挙動を，逐次的な方法で記述する方法を学びました。つまり，逐次型プログラムの作成方法を学びました。プログラミングは，単に記述です。まだ存在していないものを記述するだけです。たまたま我々は非常に限定されたき記述方法しか持っておらず，それが逐次型プログラミングだったということです。したがって，それよりも下で進行するものあると考えるのは自然です。それはたとえば，下で進行することや上で進行することを記述するオートマトン理論ですが，これらも当然異なっています。この場合，単なる散文かもしれませんし，または，より構造化された物語かもしれませんが，それは異なる形式をとります。すでにプログラミング・レベルで何か特殊なものに関わってしまっているのですから，これらのすべてのレベルの記述を持つことは，特に奇妙なこととも言えないでょう。では，それらをどのように単一化すればよいのでしょうか。これを行うにあたって，並列性に関する統一した理論を探究すると言うと，同じことを繰り返して言っていることになるかもしれません。もうひとつ私が言いたいのは，通信と並列性とは，本来同じ問題だということです。通信のない並列性はつまらないし，並列性のない通信は存在できません。つまり，これら2つのものは，全く同じなのです。

次に，基本要素について，話を進めるには，次のような考え方を避けて通ることはできないと思います。つまり，たとえば，我々がよく知っている共有メニモ・モデルやメッセージ交信モデル等，いろいろな基本要素を通信に使うとしましょう（メッセージは識別できるのに対して，メモリ中の値には識別できず，何度でも読み出したり，送信したりできるので，これらはもちろん異なっていますが）。そうすると，その世界には，必然的に2種類のオブジェクトが存在することになります。送信側と受信側があり，アクティブなオブジェクトがあり，そしてメッセージやレジスタ等の仲介者中間のものがあります。また，これら異なる種類のオブジェクトの間を結ぶ矢印があります。つまり，その世界のオブジェクトを1個だけにしようとすると，私の知る限りでは，基本要素として同期通信以外何もないことになります。さて，次のことには人は即座に反論するでしょう。特に私が押しつけるような言い方をすると，反論が来るのでしょうが，それこそ私がこれを強調する理由のひとつです。普通は自信がないと，押しつけるような言い方になるのですが，私の場合，このやり方でうまく行っています。それに，実際，我々が求めているのはオブジェクト指向モデルだ，と言う以外の何ものでもありません。「オブジェクト指向」という言葉は，その意味のあいまいさにより，便利な言葉とされていますが，たったひとつのものの部分ではなく，独立して存在し，互いに反発し合うものを意味しがちです。

したがって，プログラムもレジスタも，すべてオブジェクトです。そして，これにより，驚くほどスムーズなモデルができます。実を言えば，長年私がやってきたことは，Carlのアクター理論の数学的裏づけです。

三番目にお話しするのは，構造，これは，非常に内容豊かな課題です。それによって，大変多くのものを示すことができるからです。ここで我々は，Petriネット理論を超えることができます。Petriのネット理論は，疑問の余地なく，並列性に関する最初の真の理論であり，すなわち我々が現在把握しているような構造の取扱い方を探求していました。実際には，あたかも同一のものであるかのように見える２種類の異なる構造があります。ひとつはシステムの物理構造です。これは平板で，異なる構造から成る，つまりそれがどのように内部結合しているかに関するものです。これは仮想システムと実システムの違いとなります。しかし，もう片方のものは，それよりもはるかにつかみどころのない構造です。それは，システムを理解するために，システムに与える構造です。並列動作を行っている異機種混合システムを理解しようとする人なら誰でもわかることですが，ある種の方法で理解しようとする時，別の理解の方法の要素と関係づけることにより，うまくシステムを分析することができます。それで，我々は，ある方法で下位システムに分解することにより，構造を与えます。これこそ，我々の理論で可能にしなければならないことです。

構造には，さらに２つの側面があります。ひとつは，実システムと同程度に並列理論でカバーされなければならない仮想システムでは，それらは成長したり縮んだりします。並列プログラミング言語におけるプロシージャの活性化は，非常に理解し難いものですが，その一例です。しかし，そういった事を意味論的に理解できれば，大きなパワーを発揮できるでしょう。

しかし，それよりも捕らえにくいことは…これこそ，長年Carlがアクター・モデルにおいて研究してきた問題ですが…，エージェントがその接続関係を変えるという点です。これは，成長・縮小とは異なります。成長したり縮小する間は，近隣の構造はそのまま組持し，ただそれらの内部で新しい近隣構造を持つ新要素を作り上げるだけで，既存の近隣構造は変更しないのです。

接続関係の変更に関し，かなり微妙に異なる例を２つあげましょう。ひとつは，オペレーティング・システム中を流れるジョブです。これは，ソフトウェアと考えることができます。そこで，プロセスとプロセッサのと間の変移する結びつきを考えることができます。ここでモデルに関する難関のひとつは，ハードウェアとソフトウェアを同時に記述できるというだけでなく，プロセスとプロセッサ，さらに，そのプロセスがそのプロセッサ上で走っているという事実を示すプロセスとプロセッサの結びつきを，全て同じ方法で表現するということです。これが我々の唯一の目標というわけではありませんが，それは非常に興味深く難しい問題なので，この理論で可能ならば，他のいろいろなことも，たぶんできるのではないかと思われます。

これで私の話したいことは終わりです。

並列性に関する適切な理論についての３つのテーマ。第一は普遍性。第二は適切な基本要素を見つけること。第三は構造です。これらの３つの事柄について，必ずしもユニークな解答が見つかるとは限りません。ここでは，これで終わりにします。

座長：どなたかご意見があれば…。

HEWITT：Robinと私は何年もこれを論じています。しかし，今だに私は同期通信について悩んでいるのです。私が思うに，同期通信は規約によってのみ可能です。つまり，同期通信を行えば，同一事象に２つの異なる名称を持た

せなければなりません。たとえば，私がこの部屋を出るとして，部屋を出るということと外のホールに入るということを同期させるには，外のホールに入るということが，部屋から出るということと同じ名称でなければなりません。これは名称は２つですが，同一のイベントだからです。しかし，そこで私は，Robinのレジスタの使い方が気になるのです。たとえば，私がここにいるプログラマで，レジスタは向こう側にあるとします。すると，２つの事が起こります。ひとつは，私がここからメッセージを押し出します。そして，それが向こう側に到着するという第二の事象が発生します。この場合，同一の事象について２つの名称を持っているようには思えませんし，あるいは，それは同期とは呼べないような気がします。

MILNER：その場合，私の答えは，それらは同一事象ではないが，最初の事象と次の事象の両方に加わる中間媒体があるということになります。それだけです。この点には同意してもられると思いますが。

座長：他にどなたか…。

質問：同じ系統の問題なのですが，同期通信モデルということになると，基本的な問題は，無制限(unbounded)バッファをつくるエージェントを動的に生成できるかどうかということです。同期通信ということを考えないならば，無制限バッファや無制限となりうるバッファを持たない限り，通信の運搬が保証できる経路問題などの問題が生じます。

MILNER：そうです。ですから，実行中に新しいエージェントを生成する必要があります。その通りです。

質問：そして，同じような保証を与えるため，生成できる数を無制限にしないこと。

MILNER：確かに私が扱ってきたモデルでは，常に新しいエージェントの生成が可能でした。そして，エージェントの接続関係の変移については触れませんでした。これは別の問題です。自分が熟知している方法で，この数学問題と取り組みたかったのです。ゆっくりと進歩してきましたが，やがては移動性という考え方に研究を進めることも可能でしょう。あなたの指摘は完全に認めます。ただ，そうなると…

質問：その通りです。それは，たとえばCSPとCCSのようなもののパワーの間の根本的相違だと思います。これらのエージェントを動的に生成できてしまえば，同期通信という意味で，これらの同期通信を定義しバッファリングすることが可能です。しかし，無制限数のエージェントを生成できないと…ある意味で充分に速くということですが…そうすると，これら２種のシステムのパワーは非常に異なってきます。

MILNER：その通りです。

座長：Bill,何かコメントは？

Dally：この同期相互作用は，メッセージ交信における状態隠蔽や共有メモリにおける通信隠蔽なしに，メッセージ交信と共有メモリの両方を可能にするので，これを基本的な記述要素としようという考えのようですね。これはマシンの構築にも使用できる記述方法でしょうか。それとも，この２つのものは全く違うとお考えですか。

MILNER：これはありがたい質問です。というのは，概念モデル作成のための基本要素は，必ずしもマシン構築用の基本要素と一致しないからです。関数的アプリケーションという考え方が，概念モデルのためのすばらしい基本要素であることを示すプログラミング例は，多数あります。しかし，関数的アプリケーションをインプリメントしてみると，概念モデルにおいて環境的なものに変化を起こマシンにおいては，それが基本要素からはほど遠いことがわかるのです。それは理解できることだと思います。要するに，我々の概念マシンは長い間発達を続けてきたため，我々にとっては基本的であるもの

が，マシンにとっては，実は基本的ではない，ということなのかもしれません。

したがって，これで話が終わりになるという訳ではありませんが，この例においては，うまい一致点があるのではないかと思います。これに同意していただけるかどうかはわかりませんが。つまり，同期通信は概念モデルとマシン・モデルの双方にとり，有効な基本的概念であるということです。概念モデルと同様に，ハードウェアにも有効であるという点に同意していただけますか。

Dally：ハードウェアにおいては，Carlが説明した理由では難しいと思います。ワイヤを飛び越えるため，状態を隠してしまったり，そういう類のことが発生しがちなので。ですから，メッセージ中に状態を隠しておけるようなモデルで，同期通信と取り組んだ方が簡単でしょう。

MILNER：なるほど。そうすると，これは基本要素の不適合のもうひとつの例ということですね。

座長：この点については，後ほどあらためてということで，上田さんの発表に移りましょう。

上田：FGCSプロジェクトにおける核言語の役割についてお話したいと思います。

FGCSプロジェクトの顕著な特徴は，ミドル・アウトに研究開発を進めるとういう方法を取ったことです。つまり，一方では，核言語からハードウェアへと研究をすすめ，他方では，核言語からアプリケーション・ソフトウェアへと，研究を進めてきました。この取り組み方は，プロジェクト開始時に，仮説として採用したものですが，ここでは我々がなぜこのようなアプローチをとったかについて，説明します。

プロジェクトの初期では論理プログラミングを核言語の枠組みとして最良のものと仮定し，PrologをベースとしたKL0を設計しました。現在の中期段階では，基本言語GHCとその実用版であるKL1を完成しています。では，この核言語層の役割は何でしょうか。我々にとってこれは単なるプログラミング言語ではありません。単なるプログラミング言語であれば，並列インプリメンテーションやアプリケーション・ソフトウェアに適したどんなパワフルな言語でも設計できるでしょう。

しかし，核言語は，このミドル・アウトの研究開発の出発点なので，他の層に依存せずに，それだけで成り立つものでなければなりません。そこで，核言語には堅固な原則や優れた理論による裏打ちが必要です。これが，私がここで強調したいことです。指針の一例としては，簡素性あるいは抽象性が考えられます。

核言語は，並列プログラミングに関するさまざまな概念や要求を明らかにするための思考の枠組みを提供しなければなりません。また，この層は理論と実際の調和を図り，アプリケーションとハードウェアを結びつけなければなりません。核言語層は，これらの役割をすべてを担っているわけです。

我々はよく並列プログラミングは難しいかと聞かれますが，この質問に対する技術的にちゃんとした答えは，並列型ソフトウェアの作成にもっともっと努力を傾けてからでないと出せないでしょう。これまでに払われた努力とは，逐次型ソフトウェアの作成のために払われた努力に比べれば，ごくわずかにすぎません。我々は，フォンノイマン型コンピュータの逐次プログラミングに慣れすぎて，このことが並列プログラミングへの努力を実行に移すことを阻む障害となっています。特に，プログラムの正しさと効率を保証するのに，逐次性に頼ってしまうという点が問題です。また，よく使われる定時間アクセスの仮定は，プログラムの局部性を考慮し利用することの障害になっています。

こういった問題を解決するために考えられる手がかりとしては，使いやすい並列言語とその効率良いインプリメンテーションを提供すると

いうやり方があります。これがまさに，今我々が行おうとしていることです。我々の目指すプロセス指向プログラミングは，オブジェクト指向プログラミングと，非常によく似ているので，オブジェクト指向プログラミングの成功には非常に励まされています。さて実際に核言語を設計しなければならないのですが，それには二つの方法が考えられます。逐次言語を拡張する方法と，本質的に並列の言語を開発する方法です。この表は，これら二つの方法を比較したものです。第一の方法では，制御がover specificになり得ますが，第二の方法では，本質的でない逐次性を指向することはできません。次に，言語が粒度を定めるという点で，逐次言語を拡張した並列言語には，柔軟性が欠けています。一方，本質的に並列の言語で作成されたプログラムは，どんな潜在的な並列性も表現します。プログラマの観点から言えば，第一の方法ではプログラマが逐次性に依存できてしまうのに対し，第二の方法では言語が思考の転換を促します。したがって，長い目で見れば，後者の方法の見込みがあると思います。

もうひとつ考えなければならないことは，制御つきと制御なしのどちらかの言語を選択するかという点です。ここでは，「制御」という言葉は，効率のための制御ではなく，プログラムの正さを保証するための制御を意味します。この二種類の言語の相違は，抽象度というよりも形式化の問題で，たとえばGHCは非常に抽象性の高い言語ですが，制御の概念をもっています。これら二種の言語は，異なるレベルで異なる目的のために使われることになるでしょう。

これら二種の言語がどのような関係を持つかを見てみましょう。この二つの図は，Carl Hewittが使った図とよく似ています。左の図は，よくみかける状況を表したものです。この内側の枠の中には，制御の概念のない優れた言語や優れた理論があるかもしれません。しかし，我々が生きているのはこの外側の世界で，内側の美しい言語は，より低レベルの逐次型言語を使って，何とかしてインプリメントしなければなりません。今のところ，この内側の枠と外界との間のギャップを埋めるための原理は，ほとんど何もありません。我々は，この左の図を右の図に構築し直そうとしています。そのためには，内側の枠組みと外界とのインタフェースをとるための優れた原理と言語を見つける必要があります。このギャップを埋める優れた言語を設計するにあたり，左の図に示されている現状を綿密に再検討する必要があります。一例をあげますと，Prologシステムの使われ方の現状にはいろいろな問題があります。Prologでは，論理に無関係な機能を多用しますが，そういった機能は，純Prologの層を取り囲む優れた言語を設計することにより，考え直し，再構成すべきです。

難しい問題が与えられたときは，まず，さまざまな概念を分離しようというのが私の一般方針です。今の例では，まず内層と外層の違いを明確にし，その後，これらを再統合するのです。ただし，これは非常に慎重に行わねばなりません。

このスライドは，並列性を露出するやり方を採用した理由を記入したものです。ひとつには，コンカレント・システムの開発は，さまざまな層に関する研究に取り組んでいる多くの人々の支援を受けるべきだと考えるからです。並列性は非常に難しい問題なので，コンピュータ・サイエンスの中の限られた分野の少数の研究者に任せてしまうわけにはいきません。

並列性を見せるもうひとつの理由は，並列コンピュータを効率よく利用するには，優れた並列アルゴリズムの蓄積が必要だということです。現在の逐次型計算の文化が逐次アルゴリズムの蓄積に支えられていることを考えれば，この点は明らかです。そこでアプリケーション・プログラマも並列性にアクセスできるようにすべき

だと考えるわけです。どのプログラマも並列プログラミングをやれと言っているわけではありません。そうしたい場合に，できるようにしておくべきだ，と言っているのです。我々のやり方は，並列性をある層に閉じ込めるのではなく，単純で抽象的な形式で露出させることにより，並列性の利用を促すというアプローチです。

最後の２枚のスライドは，GHCとKL1の概要です。これら二種の言語は，実は我々の並列核言語の理論および実用版です。前期に設計した我々の最初の核言語は，Prologをベースとするkl0でした。それは逐次言語で，閉じたシステムの記述に適していました。しかし，オペレーティング・システムSIMPOSを作成するためにPrologにいろいろアドホックな拡張を施す必要がありました。中期では，GHCを本質的に並列で，外界と相互作用する開いたシステムの記述に適するように設計しました。しかし，我々はさらに次の段階に進まなければなりません。我々の核言語の理論版，つまりGHCですが，これには並列実行の観測と制御を行うために必要なメタレベルの操作が含まれていないためです。このため，核言語の実用版であるKL1には，オペレーティング・システムPIMOS作成のため，急場しのぎの方法で，メタレベル機能を追加しなければなりませんでした。次の段階では，もっとエレガントな手法でメタレベル操作とリフレクション機能を再構築しようと計画しています。

核言語としてGHCとKL1を選択した理由は，GHCが非常に原始的な言語で，さまざまな見方が可能だからです。まず，GHCはプロセス記述言語と見ることができます。しかし，GHCは非常に原始的なため，通信チャネルでさえ，言語組込のコントラクトではなく，その言語でプログラムされます。また，この言語は，データフロー言語としても，並列アセンブリ言語としても見ることができます。GHCの設定目標は通常の意味での論理型プログラミングとあまり関係ないのです。それでもGHCは，宣言的解釈が可能な論理プログラミング言語と見ることもできます。

単純であることは強固・堅牢であることにつながるので，私は核言語の中核部分を，できる限り単純にしておきたいと思っています。単純さは，並列計算の本質を顕在化させます。また，単純さは，異なる概念の区別を助けてもくれます。Shapiro博士が言うように，GHCはシステム・プログラミング等のアプリケーションには，単純すぎるかもしれません。しかし，これは，種々の並列論理型言語を比較するための基準点として使うことができます。核言語に関しては考えなければならないことがあまりに多くて一度には考えきれないので，いまは基本的な事柄に集中しているのです。すでに述べたように，GHCとKL1の間にはまだギャップがありますが，純PrologとKL0の間のギャップに比べれば，はるかに小さなものです。

単純さに関するもうひとつのポイントは，使いものになる形式的意味論が与えられるということで，現在，私はその研究を行っています。また，単純さにより，プログラムの解析・変換・最適化のための形式的な方法の開発が容易になります。これらの理由で，我々はいまだに核言語の２つの版，理論版と実用版の使っているのです。

座長：パネリストの皆さん，または会場から，何かコメントや質問はありませんか。ある方はどうぞマイクまでお進み下さい。

高山：私はフォンノイマン型の思考法よりも，数学的思考法に慣れすぎてしまっているのかもしれません。数学的思考法は並列性とはほとんど接点がありません。GHCは数学的思考法に強い影響力を持つのでしょうか。たとえば，数学的思考法に並列性を持ち込むというようなことですが…。

上田：つまり，並列プログラムの構築に，数学的思考法が役に立つか，ということですか。

高山：いや，そうではなくて，数学的思考法は非常に限られたもので，何ら並列性を備えていません。一種の並列性を備えたより広範囲な思考法へと，それを拡張できれば，と思うのです。並列性を備えた理論へと数学的思考法を拡張することにGHCは強い影響力を持つのでしょうか。または，何か良いヒントを与えてくれるでしょうか。

上田：ここで私が言えることは，並列性の問題は，数学的思考法にとっては非常に新しい考え方だというとぐらいですね。

座長：これは全体討論で続けられるかもしれません。次の発表に移りましょう。

WARREN：いろいろな意見を聞いていると，２つの概念がごちゃごちゃになっているような気がします。他に適当な言葉もないので，これらの概念「パラレリズム」と「コンカレンシー」と呼ぶことにします。

「パラレリズム」と言う場合，計算速度を実際に向上させるという目的で，同時にいくつもの活動を実行するコンピュータを念頭に置いています。そして，「コンカレンシー」を表現するというもうひとつの立場があり，こちらでは，オブジェクト間，プロセス間の通信という考え方が，アプリケーションにとって本質的な意味を持つようなアプリケーションに注目しています。たとえば，オペレーティング・システムの設計，複合プロセスのシミュレーション，航空券予約システム等々がこれにあたります。つまり，アプリケーションでの並列性，コンピュータでの並列性，そのどちらについて考えるかという問題なのです。そして，これらは，まったく異なるものです。ですから，たとえば，逐次型コンピュータ上で走る並列型（コンカレント）アプリケーションもあり得るわけです。実際，航空券予約システム等のほとんどの並列型（コンカレント）アプリケーションは，現在，大型逐次コンピュータ上で稼働しています。また，原則としては，並列型（コンカレント）アプリケーションを「パラレル」コンピュータ上で稼働させることも可能です。しかし，普通の非並列型アプリケーションでも，逐次型あるいは「パラレル」のコンピュータ上で，同じようにうまく稼働させることができるのです。したがって，この討論に関し，これら２種の動機を区別することが重要だと思います。

私個人の最大の関心の的は，「コンカレンシー」よりは，「パラレリズム」の探求にあります。つまり，「パラレル」コンピュータ上での計算速度の向上です。そして，重要なことは…これは上田さんとは逆の立場になりますが…パラレリズムを，普通のプログラマから見えないようにすることです。複雑なアプリケーションについてだけでなく，コンピュータ内部で進行しているパラレリズムにまで，プログラマが気をつかわなくてはならないような状況をわざわざ作らなくとも，すでにプログラミング自体，非常に難しい作業です。

理想的には，パラレリズムをプログラマから見えないようにすべきです。実際，今日我々が通常逐次型マシンだと思っているものでも，最低レベルでは，しばしばパラレルになっているのです。たとえば，逐次型マシンでのパイプラインでは，最低レベルである程度のパラレリズムが存在していますが，ただそれは通常，最低レベルの命令セットを解釈するマイクロプログラムを開発するマイクロプログラマにしか影響を及ぼさないのです。そして，それよりも上のレベルからコンピュータを見る人は皆，必ずしもマシンのパラレリズムに気つく必要がないのです。

たとえ大規模パラレル・マルチプロセッサについて語るにしても，どんなパラレル・マシンにおいても，その程度の区別はしたいと思うの

です。そこでプログラマは，ビルを建設する建築家のような存在であるべきだと思います。建築家はビルの大きさと形を指定し，コストの概算をしますが，ビルをどのように建設するかの詳細には，配慮する必要がありません。その仕事は建築会社の受け持ちで，コンピュータ・システムはこの建築会社のようなものです。仕事をできる限り速くすませるためには，どれだけのリソースを並行して動作させればよいかの決定には，この底辺のシステムのみが責任を負います。

さて，さまざまな言語に関し，逐次型と並列型のどちらが必要か，それとも他のものか，という話が出ました。私は言語を，逐次型，並列型，宣言型の３つのカテゴリーに分類すべきだと思います。

逐次型言語は従来のフォンノイマン型言語です。並列型言語は，それより新しいけれども，やはり従来の言語で，パラレル・マシンのプログラミング用に作られています。これらの言語においては，マシンの中で起きていることについて考えます。オペレーションを割り当てることにより，状態をいう概念，そして，状態の変化という概念が，これらの言語に組み込まれています。逐次型と並列型の違いはと言えば，逐次型言語では，状態は一度に一段階ずつ変化し，並列型言語では，一度に多くの場所で状態が変化し得るということです。しかし，基本的には，ここでは底辺つまりマシン中で進行することをいかに指定するかという観点から，プログラムを考えています。

それとは異なり，宣言型言語は，もっと上のアプリケーション側を見ています。つまり，何をしようとしているのか。何が問題なのか。その問題をうまく記述し，コンピュータ上で解決するにはどうすればいのか。宣言型言語の例は，Prolog,GHC,その他ユミッテッド・チョイス型の言語です。これらの言語はいずれも問題を宣言的方法で記述します。コンカレント・アプリケーション，たとえばオペレーティング・システムを見てみると，これらのアプリケーションには，GHCのような言語が，きわめて有効であることが証明されています。一方，アプリケーションにとってコンカレンシーが本質的でないようなアプリケーションを見てみると，そこではPrologの方が適しています。

この討論の準備の際，座長が我々に議題として示したのは，未来の「並列*」コンピュータ・システムの姿という点でした。そこで，最後に今後マシンにどうなって欲しいかという私の考えを述べたいと思います。

*脚注：「パラレル」コンピュータ・システムも含む。

底辺の実際のハードウェアから始めると，昨日の発表で，データ拡散マシンというものの説明をしました。その内容を簡単に紹介しましょう。その狙いは汎用マルチプロセッサ・アーキテクチャの構築でした。それは任意のサイズに適用が可能で，また，グローバル・アドレス・フェーズあるいは共有仮想メモリという概念をサポートする必要があります。しかし，マシンは分散した物理メモリを備えるという点で，メッセージパッシングマシンに似ています。また，マシンは，Herb Simonがハードウェアはかくあるべきと言った姿と一致する階層型マシンです。このマシンは論理型プログラムの実行という目的を念頭に作られてはいるものの，完全に汎用マシンです。実際，どんな種類のソフトウェア・アプリケーションに関しても，同じようにうまく使えるはずです。この図は，データ拡散マシン上で稼働するソフトウェアに落ちる非論理型プログラミング・アプリケーションの例です。実際，今日，SequentやEncore等の従来の共用メモリ・マシンは，データ拡散マシン上でも同じようにうまく稼働するソフトウェアを備えています。これが汎用マシンだという

のは，このためです。

しかし，私個人の関心は，並列推論マシンの開発にあります。我々は，何らかの適切な基本的ソフトウェアを供給することにより，データ拡散マシンを並列推論マシンに変えようとしています。その基本ソフトウェアに関する現在の概念は，Andorraモデルと呼ばれるものです。これは午前中のSeif Haridiの発表で，簡単に触れていました。

Andorraモデルの概念は，OR並列性と，それに依存するかあるいはストリームのAND並列性を組み合わせることです。実は，このAndorraモデルは，PrologのOR並列性インプリメンテーションであるAuroraシステムで行った研究を，拡張するための概念です。そのインプリメンテーションは，逐次型マシンで走らせる場合と同じ文法で，標準言語Prologを並列型マシン上で走らせるためのものです。

データ拡散マシンあるいは既存の並列型ハードウェア等のハードウェア上でAndorraモデルが動作する場合，原理上，それは並列推論マシンであると言うことができます。この並列推論マシンの上でアプリケーションをプログラミングするには，どんな言語が適切でしょうか。ここで私が強調したいのは，ユーザに関する限り，並列推論マシンはブラックボックスで，マシン中にパラレリズムが存在するという事実を，ユーザは認識しないということです。そこで，アプリケーションに適切で，その下にあるパラレリズムをユーザから隠すような言語が望ましいことになります。私がAndorra Prologと呼んでいるこのタイプの言語は，まだ発達段階にあります。これはAndorraモデルつまり底辺の計算メカニズムを利用する言語です。この言語は，Prologと，GHCのようなユミッテット・チョイス型の言語の両方を包含します。

したがって，この言語は，現在Prologで作成されているアプリケーションの実行に適していると思います。それらを直接Andorra Prologにマッピングし，前と同じように実行することができます。同様に，現在GHCのようなユミッテット・チョイス型の言語で作成されているアプリケーションもAndorra Prologにマッピングし，逐次型あるいは並列型マシン上で，同じようにうまく稼働させることができます。底辺のマシンがプロセッサを一つしか持たない逐次型か，それとも並列型かは，ユーザにはわかりません。

しかし，Andorra Prologは，PrologやGHCよりも汎用性が高いのです。そして，午前中にSeif Haridiが述べたことですが，興味深い点として，Prologにもユミッテット・チョイス型の言語にもマッピングが難しいアプリケーションでも，Andorra Prologでは可能だという事実があります。

そろそろ時間切れですので，ここまでにします。

座長：ここで進行している知的闘いの性質が明らかになってきましたね。誰もが上に行きたがる相撲と違って，ここでは皆が底辺レベルに行きたがり，ライバルを上に置いておきたいようです。何か質問は？

DALLY：２つ質問があります。最初はLISPでよく言われることで，LISPのプログラマはあらゆるものの価値を知っているが，コストについては何も知らない，という言い方をします。並列マシンでは，コストのほとんどは通信に関してで，アプリケーションの性能は，アルゴリズムがいかにうまくパラレリズムを利用するかという点にかかっています。パラレリズムをプログラマから隠すのはナンセンスだと思いますが。

WARREN：パラレリズムを見えなくするということは，大型アプリケーションを稼働させるための理想的状態です。たとえば，何か複雑な自然言語システムを開発する場合，システ

ムが大きくなるほど，開発者はアプリケーションの正しいモデリングに気をつかうようになります。並列マシン上でより速く動かすことができれば，確かにその方が良いでしょう。しかし，最も気を使わなければならないのは，問題を正しく表現し，解決することです。ですから，このようなタイプのアプリケーションに関しては，パラレリズムに関わって作業を難しくすることは避けるべきでしょう。

ただし，あなたや上田さんが言うように，今のところ，問題を並列アルゴリズムにマッピングする方法はわかっていません。したがって，短期的には，おそらくマシン中でのパラレリズム利用方を制御できる方が望ましいでしょう。しかし，これは長期的目標ではありません。今後起きることとしては，底辺のパラレリズムの利用方を多種多様なアノテーションにより言語に対して直接指定するということが考えられます。GHCをベースとした言語でICOTが行おうとしているのはこのことです。つまり，この計算がどこで行われるかを，実際に物理的に制御するのです。それは理想的とは言えません。パラレリズムの利用法がもっとよくわかってくれば，その知識すべてを底辺のシステムに埋め込み，それにより，システムがその世話をしてくれるようにできるのではないでしょうか。そうすれば，アプリケーション・プログラマは，このようなことに気を使わずにすみます。

DALLY：次はプリミティブについての質問です。Robinは，まず最初に考えるべき適切なプリミティブとして，関数呼出しをあげていましたが，今日のコンパイラ作成者は，マシン中に関数呼出し命令があることをいやがります。同様の考え方として，共用メモリ参照は，それについて考えるのは面白いのですが，そのインプリメントはたいへんです。共用メモリ参照がハードウェアプリミティブでなければならない理由はあるのでしょうか。

WARREN：ハードウェアプリミティブであろうがファームウェアプリミティブであろうが，基本アプリケーションに関しては，プリミティブにする必要があります。なぜなら，並列計算を考える場合，さまざまな処理エージェント，データの共用，データの取扱い，という観点から見る方が自然です。その際，各エージェントは一度に一段階ずつ作業を行い，問題のデータ全体は，一度に数段階ずつ処理されます。データの共用という考え方は，実際には共用仮想メモリの考え方と一種共通しており，データの名前としては仮想アドレスしか見えません。優れた並列型ソフトウェアの作成には，共用仮想メモリという概念が非常に重要だと私が思うのは，このような理由からです。

したがって，直接ハードウェア中にインプリメントされるか，ファームウェア中にインプリメントされるかは，問いません。ただ，共用仮想メモリをサポートするインタフェースは必要でしょう。

HEWITT：どういう意味合いでGHCを宣言的と見做すのですか。

WARREN：宣言的言語という概念の意味するところでは，プログラムを作っているステートメントの宣言上の意味という観点から，プログラムの結果を考えることができる，ということです。GHCやユミッテット・チョイス型の言語では，プログラムは永久（パーペチュアル）的なプロセスになりがちで，したがって，実際には結果を生みません。それらは時と共に発達します。しかし，中間段階での仮定に対する結論を重ねることで，時間がたつにつれ，それらの行っていることがわかってきます。この意味で，そのようなプログラムはそれらを作っている節の宣言内容を，正しく反映していると考えることができます。しかし，論理型プログラミングでは，論理型プログラミング言語は単に宣言部分だけでなく，制御部分でもあり，そ

のように使われるということを認識することは重要です。したがって，単にプログラムの宣言内容のみを理解しても，全部わかるわけではありません。プログラムの制御部分の推論も必要です。制御部分を除き，宣言部分のみの推論ですめば楽なのですが，実際にはそれではすみません。うまく推論できるような形の制御も必要です。

座長：他にコメントは？

MILNER：プログラムの表現からプログラムの結果がはっりきりわかるということは，必然的に，それが論理的に推論できるということを意味するということを言っておられたようですが，私にはその仮定は，疑問なのですが。いろいろな数字がありますが，プログラムの記号，論理的以外の数学的方法に分析した結果と，論理的推論の結果との間には，何ら質的な相違がありません。そして，これら2つのものは，完全に同等に見えます。論理型のプログラミングが何らかの優位性を持つとは思えないのです。

WARREN：私が言えることは，論理型プログラムを見ると，プログラムの宣言内容を使い，プログラムを部分ごとに理解することが容易だということです。また，数学的オブジェクト全体の推論の試みるよりも，ひとつの小さな節を見て，その節の意味を知ることの方が容易です。これが，論理型プログラミングにあり，他の形式方法には無いものです。プログラムの小さな部分を見て，個々の部分が正確であることを確かめ，それにより，全体が正確であるに違いないと結論できるということです。

MILNER：並列性は非常に新しい問題なので，その可能性はあるでしょう。そして，並列性に関して正しい数学的研究を行えば，現在あなたが論理に対して抱いているような自然な感じ方ができるようになるかもしれません。その結果，ある意味で明快ではあるが論理的でないプログラミング概念を得ることができるかもしれません。たぶん，我々の側の訓練の問題なのでしょう。

WARREN：並列型（コンカレント）アプリケーションに論理型プログラミングを使った場合，真に宣言的言語透過性は，Prologのような言語に関してよりも，プログラムの理解全体を包含する程度が低いと言えます。したがって，並列型（コンカレント）アプリケーションについて考える場合は，非並列型アプリケーションについて考える場合よりも，制御に対する考察が必要です。

上田：Hewitt教授と私は，彼の言うところのマイクロ理論の記述にはPrologが，そして，開いたシステムの記述にはGHCが適していると考えています。Prologは優れた，しかも単純なセマンティックスを備えています。GHCもかなり単純な意味論を備えています。しかし，これらの言語の意味論は，簡単に組み合わせることができるとは思いません。Andorra Prologでは，意味論はどのように組み合わせるのでしょうか。統合型言語のための意味論はありますか。

WARREN：Andorra Prologでどのように意味論を組み合わせるかということですか。Andorra Prologはまだ完全に定義できていません。しかし，宣言上の観点から言えば，節は節です。節が意味するところはわかっています。演算上の観点から言うと，Andorra Prologを詳しく説明しない限り，その説明は難しいと思います。しかし，Prolog系列のアプリケーションを作成しようとすれば，演算上のセマンティックスは，Prologとよく似ていることがわかります。それが並列型のアプリケーションの作成であれば，必要な推論はGHCの場合と非常に似ていることがわかるはずです。次に不確定性とコルーティングの両方を含むアプリケーションであれば，その両方で並列性となります。うまく答えられなくてすみません。

HEWITT：ちょっと待って下さい。FGHCで私が妙だなと思うのは，局地的状態が時々刻々変化する場合の共有オブジェクトをどう説明するか，という点です。FGHCに共有口座を持っているといましょう。それをどう扱えばいいのでしょうか。金を引き出そうとして，その口座へのメッセージを用意します。そこで，ストリームの最初のエレメントはWITHDRAWというメッセージで，それには成功か失敗かを記録するための書込み専用変数が付いていることをアサートします。これは手続き型言語でプロシージャを呼び出すようなメカニズムです。結果が戻ってきます。金を獲得できたかできなかったのかのどちらかです。どちらになるかは，論理的に演繹できません。私だけでなく，他の誰にもそれは不可能です。したがって，それは非常に演算的に見えます。この口座の使い方を考えるとWITHDRAWメッセージが口座からの応答という変数に対するメッセージ・ストリームの最初のエレメントであることをアサートします。応答を待っていると，やがて，「金を獲得した」か「しなかった」という答えを得ます。この答えは，私が行ったことの論理的帰結ではありません。これは演繹の制御という問題ではなく，充分な演繹を行えないことが問題なのです。

WARREN：私が思うに，それは，実行したことの帰結として何かが起きるかどうかを発見しようとする場合，必要なものは論理だけではないからです。論理型プログラムは一般に，論理プラス制御でできているからです。この場合，制御はコンカレンシーを制御しているのですが，その制御言語が何をしているかを知ることが重要です。しかし，その質問に関しては，上田さんがShapiroに説明してもらった方がいいと思います。

座長：では，これを続けましょう。まだDavidに対するコメントがあるようですので。言語を逐次型，並列型，宣言型に分類するというやり方は，たいへん有用だと思います。しかし，GHCやその他の並列論理型言語に関するあなたの分類は，誤っているのではないでしょうか。それらはOCCAM,CSP,等々と同じ仲間です。これらも，宣言型言語よりもはるかに複雑で異なる種類のものでたまたま宣言読取りをするのです。並列論理型言語の宣言読取りは，次のようなものです。「以下があるプロセスで可能なヒストリの列であれば，これもプロセスの別のヒストリである。」しかし，並列論理型プログラムにおける宣言推論は，単純な論理型プログラムにおけるような値間の論理的関係についてではなく，プロセスのヒストリについて行われます。したがって，その分類は確かに有用かもしれませんが，GHCは宣言型よりも並列型に入れるべきです。それは，GHCを理解するためには，制御面に関する推論を行う必要があるからです。また，これはCarl Hewittの質問にも答えることになりますが，これらの言語を宣言的に考えたくなければ，それはそれでかまわないのです。これらの言語に対しては，適切な操作的意味論があります。それについてのみ考え，論理型プログラミング面は，忘れてしまってかまいません。

ポイントは，我々の多くが，操作的な面に加え，論理型プログラミング的な面について考えれば便利だと考えているということです。しかし，そうしたくなければ，それで良い。選択の余地があるのです。

会場からコメントがあるようですね。

Seif Haridi：Andorra Prologについて一言。ちょうど今，我々はAndorra Prologの一部に対するきわめて正確な演算モデルを持っています。この部分の定義はこの会議用の論文に盛り込みましたが，Davidも言ったように，我々のシステムでは，まだ意味論の変更があるかもしれません。つまり，現在，正確な操作的

意味論はあるのですが，表示的意味論は無いとでも言いましょうか。

しかし，この操作的意味論を使い，GHCで作成しようとするようなものを何とでも作成できますし，また，PrologとGHCを容易に結びつけるようなものを作成することも可能です。Prologで作るものとGHCで作るものとは，スムーズに結びつきます。

座長：他にコメントは？

Henry Libermann：パラレリズムを隠すことの目的について話したいと思います。それは目的としてはまったく信じられないようなものです。その理由は，多くのアプリケーションでは，アプリケーションの全体目的が，コンカレンシーを見えるようにすることだからです。簡単な例としては，フライト・シミュレータのようなものを考えてみると良いでしょう。スクリーン上に飛行機が表示され，燃料と高度のゲージがいくつかあります。もちろん，ゲージ用のプログラムを自然な方法で作成し，燃料の計量とそのスクリーン上での表示，燃料の現在量をX時間おきに計量し，スクリーン上に表示する，等々を平行して行いたいでしょう。そして，もしもある言語で，これらのゲージがすべて平行して動いていることを言えないとすると，どうしてそんなプログラムを作成できるのかわかりません。したがって，パラレリズムは見えないという考え方の正当性を証明していただきたきのです。

WARREN：わかりました。私の最初のスライドがあまり明らかでなかったようです。あなたの言ったことは，私の言葉では，「パラレリズム」ではなく，「コンカレンシー」です。アプリケーションが逐次型マシン上で動作するということが目的と思われますので。私が言いたいのは，「コンカレンシー」を見えなくするということではなく，底辺部のハードウェアの「パラレリズム」を見えなくすべきだということです。わかりますか？わからない。たとえば，パイプライン・マシンについて話したように，そこでのパラレリズムは普通のプログラマには見えていません。

MILNER：ちょっと付け加えていいですか。これらの両方を備えることも可能だと思います。コンカレンシーがその問題の構造の一部であれば，コンカレンシーを表現できるはずです。それをマシン中のパラレリズムから切り離したいという皆さんの意見に賛成です。また，これらの事は，プログラム中のあるプロセスは，ある特定のプロセッサ上で動くべきだということを，ある言語中で表現してはならないということを意味しないと思います。なぜプログラマにそれが許されないのかがわかりません。それを行いたくないプログラマでも，コンカレンシーをも含めたあらゆる種類のものを表現できる能力を持つようにすべきです。ある特定のマシン上で起きていることを表現することが重大な意味を持ち，ジョブを速く走らせるためにそれが必要だという場合，その能力を付け加えてはいけないという理由がわかりません。

WARREN：まとめると，「コンカレンシー」は見せるべきで，「パラレリズム」は隠すべきだ，ということのようです。それで少しは明快になるのでは？

HEWITT：制御の問題について話したいのですが。理論だけではできないことですが，GHCに関する現在の状況を考えると，制御はこの状況から我々を救ってくれる頼もしい味方なのかもしれません。

Kowalskyが制御という概念を導入した際，木から余計な枝を切り払うということがポイントだったと思います。考えられる演繹推論の枝がたくさんあり，誰かが枝を払うなら問題は無いのです。しかし，口座中のGHCに関しては，状況はそうはなっていません。私にはストリームがあり，引き出す最初のエレメントをアサー

トし，それを下へ送ります。そして，結果を受け取ります。オーケーであることを証明する必要がある演繹推論が多すぎるとは思えません。私が得た結論は，全く演繹推論に従っていないからです。ここに問題があるようです。

座長：Carlは論理型プログラミングとその並列論理型プログラミングとの関係のことを言っているのだと思います。私も発表の準備をしてきておりますので，その質問には発表で答えます。

SHAPIRO：もう年なので，この討論の進行役としてそこに座っていればいいかと思ったのですが，他のパネリストの皆さんが，そこまでの年にはなっていないし，自分の考えを述べるべきだと言うので，発表したいと思います。

あるイメージ，つまり，将来のコンピュータ・システムの仮説的イメージを提示したいと思います。また，何がいちばん底辺にあるべきかという点に関しても考えがあります。このイメージは2種の概念から成ります。ひとつは論理的プログラミングで，もうひとつは部分評価です。この両方を説明し，これまでに述べられたこととそれらがどう関係するかを示します。

論理型プログラミングは，今だに論理型プログラミングの主役であるPrologの形で実現されました。しかし，ここ2～3年の間に，論理型プログラミングには，非常に大きく重要な発展が見られ，その結果，多様性が増し，多種多様な分野に応用されるようになりました。そのひとつは論理データベースへの応用で，MCCのLDLシステム，ECRCデータベース・システム等があります。制約論理型プログラミング分野で，Prolog3, CLP(R),CHIP等々，非常におもしろく有望な研究が行われ，これらは非常にパワフルなアプリケーション・プログラミング言語であると思われます。また，高階論理型プログラミングにおいても，知名度は低いけれども，やはり重要な研究が行われています。ここでは，Dale Miller等によるLambda Prologをあげておきましょう。さらに，この会議中でさかんに取り上げられたように，並列論理型言語であるGHC,Parlog,Concurrent Prolog等も開発されました。論理型プログラミングが進もうとしているこれからの方向の各々が非常に重要であり，他の開発から独立した独自の立場を取っています。したがって，他の研究を脅かしたり，他の研究の成功に依存しているわけではなく，非常に実りある成果へとつながる明確で，きわめて有望な研究の筋道なのです。

何年か前にわき起こった疑問点で，今もなお疑問のまま残っているのは，将来のコンピュータ・システムの基礎として，どのように論理型プログラミングを利用できるかということです。また，これらの分野すべてをどのように総合または応用するか。将来，どのように応用できるか。おそらく，もっとも普通の答えは，UNIXで稼働するワークステーションのネットワークを構築し，そのネットワーク上のアプリケーションとする，ということになるでしょう。

しかし，これはあまりうまく行くとは思えません。抽象化のあらゆるレベルに対し，ある程度のシステムの複雑さが存在し，その上に構築して行くと，システムはそれ自身の重みで崩壊してしまうからです。したがって，抽象化の層を何層も重ねない限り，将来構築しなければならない高度に複雑なシステムの構築は不可能です。パネリストの皆さんのほとんどが，システムを構築するための様々な抽象化の層を提示してくれました。私もそのひとつを説明します。

抽象化の層という考え方ときわめて密接に関係する考え方として，部分評価があります。部分評価とは何を意味するのでしょうか。これについては，二村さんがすばらしい発表をしてくれました。その概念をここで繰り返すことはしません。その意味するところのみ説明します。おそらく，もっとも重要なことは，高級言語に

は，先天的なオーバーヘッドは存在しないという点でしょう。ある解答の仕様記述に高級言語を使うと，仕様の動作が遅くなり，低級言語を使うと速くなる，というようなことは起きません。単に解答の仕様記述に高級形式を使う場合，どの特定の層に関しても，形式と解答を合わせたものの部分評価が可能です。原則として，解答中に記述したもの以上のオーバーヘッドはありません。したがって，使用する形式の表現力がもっとも重要で，その重要性は，形式が高次であろうがなかろうが，それに付随するオーバーヘッドの重要性も大きいでしょう。

もうひとつは，抽象化の層というオーバーヘッドを除去することができるということです。したがって，抽象化の層を重ねることを恐れる必要がありません。原理として，それらを除去する方法がわかっているからです。それが部分評価です。図にあるように，原則として，何らかの解答の仕様記述を行う高級言語アプリケーション・プログラム，その高級言語用のコンパイラ（アーキテクチャへのコンパイルを行う），何らかのハードウェア記述言語によるアーキテクチャのハードウェア記述を選び，次に，これら全体を部分評価して，特殊ハードウェア記述を作成します。これは，次にVLSIコンパイラを使い，ハードウェアを生成します。つまり，ある意味で，部分評価には限界がありません。任意の抽象化の層を通じて，充分巧妙に部分評価を行うことが可能です。

これらの意味から結論すると，他の有用な形式すべてを自然に，しかも効率良く埋め込むことができる抽象形式を，汎用コンピュータ・システムの基礎として採用できるでしょう。

私個人の結論としては，並列論理型言語から，そのような形式がうまれてくると思います。私の研究が，誰かが並列論理言語で発明した形式をすべて埋め込むことにとらわれている理由が，これで少しはわかってもらえるでしょう。つまり，並列論理型言語が，新旧の形式の自然で効率良い埋め込みをサポートするために充分な表現力を備えている，という確信を得たいのです。

これが可能ならば，将来のコンピュータ・システムの姿も見えてきます。底辺のハードウェアは，ワークステーション，おそらく（予見できる将来において）三次元メッシュの並列コンピュータ，データベース・マシンから成り，その上に並列論理型言語にもとづく抽象化の層とオペレーティング・システム，さらにその上に，他の言語が乗ります。並列論理型言語でプログラムしたいアプリケーションもあるでしょうが，一方，並列性をもたない制約言語でプログラムしたいアプリケーションもあります。また，関数型言語でプログラムしたいアプリケーションもあります。つまり，コンピュータ・システムで，あらゆる形式または有用な形式をサポートしたいわけです。

オーバーヘッドはどうするか。答えは，部分的に評価するということです。したがって，適切な部分評価技術があれば，アプリケーションは直接ハードウェアにマッピング可能であり，ソフトウェア中に構築する何層もの抽象化層を除去することができます。

パネリストの皆さんから，何かコメントがあれば…。

WARREN：埋め込みがどれだけ効率が良いかを正確に知るためにはどうすればよいのでしょうか。それはどれだけ効率が良く，また，その特性をどのように表せるでしょうか。

というのは，高級言語から並列言語への埋め込みは，充分効率が良いとは言えないと感じているからです。それらは，高級言語での一演算が，並列論理型言語の一回の一定時間演算としてマッピングされるという一定時間埋め込みではありません。このため，それらはタスクに対して適切とは言えません。

SHAPIRO：一定時間埋め込みは存在する，

というのが答えです。Concurrent Prologに関する本の中のある論文で，アーキテクチャに対する言語の適性のテストをとりあげました。その中で，そのような言語を提示するための基準として，一定時間と一定記憶領域のオーバーヘッド埋め込みを持つという基準が提案されています。私はそれをFCPという特定モデルについて示しましたが，この場合FGHCでも同じことです。したがって，答えは，一定時間埋め込みは可能だということです。そして，それはきちんと証明できます。ある特定の高級言語に対する効率の良い埋め込みが可能か，という点は，また別の問題です。なぜなら，前者の証明は，フォンノイマン型マシンのシミュレーションによって行われるからです。つまり，Prologコンパイラでフォンノイマン型の命令セットにコンパイルし，FGHCによってフォンノイマン型マシンのシミュレーションを行います。これにより，その証明で一定時間オーバーヘッドが得られます。しかし，実際の埋め込みを行うには，もっと巧妙な手法が必要です。たとえば，別の論文では，OR並列PrologからFlat Concurrent Prologへのコンパイルが取り上げられていますが，そこでは，OR並列Prologの並列論理型言語へのより実際的な埋め込みが示されています。これには議論の余地がありますが，実際的結果は得られるようです。

したがって，今日のインプリメンテーション技術では，妥当ではあるが，他とは，比肩しえないかもしれません。ただし，原理は必ず証明できます。そして，部分評価という概念には，限界がありません。したがって，今日それが可能であるがどうかは，部分評価の応用方法の把握度にかかっているかもしれません。しかし，この一定時間オーバーヘッド理論が証明できる限り，原則として，高級形式に先天的に付随するオーバーヘッドはありません。

WARREN：OR並列Prologとコミッティッドチョイス言語とで，それが可能だということですか。

SHAPIRO：並列論理型言語ということならば，その通りです。それは可能で，その証拠も提示しました。

WERREN：可能だということですか。それともすでになされているということですか。すでに実現された証拠と思えるものは，見たことがありませんが。

SHAPIRO：これはPrologに関してすでに行われており，また，理論的分析によれば，平衡木に関し，ある特定のインプリメンテーション技術を使用したために，逐次型インプリメンテーションと比較して，対数的オーバーヘッドが得られることがわかっています。

WARREN：それは，あるプログラムのサブセットについて，一定時間ではなく，対数的なオーバーヘッドがあるということでしょう。それは我々が求めているものとは少し違うのではないですか。

SHAPIRO：研究は進行中で，皆それぞれに違う段階で，半熟の概念と半熟の結果を持っています。ただ，この結果は，少くとも私にとっては，Andorraに応用してみるだけの確実性を備えています。実際，それは，この会議終了後に，FCPに埋め込んでみようと思っている計算モデルのひとつで，その結果については，いずれ会議か何かで討論できるでしょう。その時点で，それが現実的かどうかを言えると思います。また，たとえば，FCPでAndorraのインタプリタを作成するのは，どれくらいたいへんか，動作がどれだけ遅くなるか，あなたがAndorraのインタプリタを作成するには，どのくらいの時間がかかり，動作にはどれだけの時間がかかるか，そういったことも比較できるでしょう。

Dally(?)：システムの観点から見た場合，一定時間埋め込みがあるかどうかについては，あまり触れず，定数とは何かということを話し

ていたようです。たぶん，この場では少数意見でしょうが，私は，論理型プログラミングについては，熱心な支持者ではありません。しかし，同一化を基本機構とするほどのパワフルな言語が，はるかに原始的な方法で通信を使う言語のモデリングが可能なほど基本的かどうか，という点が疑問です。部分評価という場合，FCPのレベルよりもはるかに下のものへのコンパイルを考えていますか。

SHAPIRO：全般的には，そうです。しかし，FCP,FGHC,Prolog等の言語についても，この答えはあてはまります。Prologの成功は，我々が単一化の特殊ケースを一定時間フォンノイマン型演算にコンパイルしたことによるものです。Prologを実際に使ってみると，90％あるいは95％の時間，プログラムは一般の同一化ではなく，ロード，記録，cons,car,cdr等の単純な演算をシミュレートするための演算を使っています。そして，それらは効率良くコンパイルされるので，これらの目的で使用することのオーバーヘッドは，小さいものです。したがって，そのような広範囲な形式を持つことの利点は，パワーヘッドは，小さいものです。したがって，そのような広範囲な形式を持つことの利点は，パワー全部を利用しないサブセットを使っている限り，全パワーに付随オーバーヘッドなしにコンパイルできるということです。しかし，全パワーを使うXパーセントの例では，代償を支払うことになります。ただし，それから得られるものはわかっています。

つまり，それ以上コンパイルしなくても，言語そのままですでにインプリメントされており，単純な演算をシミュレートする演算には，一定時間，それも短い一定時間しかかかりません。また部分評価技術は，必要があれば，この抽象化層を横切るために使えると考えています。しかし，それは単なる推量です。

座長：また順番に戻りましょう

MILNER：昨日，二村さんの発表で，部分評価は，もちろん限定されたプログラム変換とも考えられ，したがって，それはコンパイルだ，ということを聞きました。効率の良いインプリメンテーションを目標とした場合，単にプログラムの部分評価という方法を，論理型プログラミング言語までと考えるのは，少し制限しすぎではないでしょうか。プログラム変換をより広くとらえ，ある特定のアプリケーションにとって非常に楽で自然な形から，ある特定のマシンにより近い別の言語への変換というように考えるべきだと思います。ここにはいろいろ未解決の問題があり，この考え方のどこかには，論理屑プログラミング言語があるはずだ，という思い込みがあります。しかし，その正当性は証明されません。この世界ではそれが必要だと仮定してもいいかもしれません。しかし，プログラミングに関する重要な観念は，しばしばある種の制限された科学によってもたらされた，と私は強く信じています。たとえば，マトリックスはプログラミングに非常に役立つ概念です。論理も非常に重要な概念で，プログラミングに非常に役立ちます。現在，我々が探しているものは（たまたま論理ほど早く発見されなかったわけですが），並列性の数学理論で，それを発見し，正しく分析した後，プログラミングに導入することができるでしょう。ある特定の数学的にエレガントな分析形式，つまり論理に束縛されてしまう経路をたどる危険性があると思います。

座長：上田さん。

上田：残念ながら，部分評価技術はオールマイティではありません。ある種の高級言語については，そのメタインタプリタを作成し，それを部分評価することは容易かもしれませんが，他の高級言語では，インタプリタよりもコンパイラの作成の方が単純かもしれません。3年前，純PrologをGHC上にインプリメントしようと

試みたことがあるのですが，直接コンパイラを作成する方がはるかに単純であることがわかりました。それはコンパイルはデータフロー解析を必要とするからです。したがって，そのようなデータフロー解析が必要ない場合に限り，メタインタプリタ方式を使えるでしょう。

SHAPIRO：その2点について答えましょう。まず，部分評価と言う場合，狭義に限定しているわけではありません。私が言いたいのは，高級記述を行うための全般的な概念の枠組みがあること，高級記述も低級記述も同等なのだから，高級であるほど代償を必要とするわけではないこと，そして，それらの間でマッピングを行う概念上の手法がある，ということです。このマッピングの方法は，実際の技術によって変わってきます。

たとえば，Logixシステムでは，良い部分評価器が無いため，マニュアルでたくさんのマッピングを行っています。

また，誰かがFGHCでLDLを書き直し，部分評価して元の形に戻す，というようなこともすすめていません。ただ，並列論理型言語は表現力があるので，充分LDLの機能性を仕様記述できます。このため，LDLのような均一なシステムに対しては，FGHCで記述したとき，そのインタフェースが，あたかもFGHCプログラムを部分評価した結果のように見え，それが残りのFGHCシステムとインタフェースするようにできます。

つまり，部分評価は統一的概念であり，今日あるいは将来の部分評価技術は，我々の進歩を妨げるものではない，ということです。同様に，Logixでは，部分評価器がなかったため，メタレベル機能をマニュアルでインプリメントするために，この概念を使うことができました。ただし，その統一的な枠組みはというのは，あらゆるものの機能性が，あたかも形式単一のによって仕様記述され，部分評価されたかのように見えるということで，実際にその通りに行われたかどうかは，別問題です。

論理レイヤの質問に関しては，これは作業仮説であり，確かに，まだ証明したわけではありません。しかし，科学との取り組み方には2種類あります。そして，私はまず大胆な推測を行い，後に否定あるいは肯定するというPopperian方式を信じています。否定された場合も，推測が大胆であるほど多くを学ぶことができます。そして，成功すれば，勝ち得るものも大きいのです。ですから，現時点では，私の仮説は大胆な推測であり，他の人々には他の大胆な推測を行ってもらってかまいません。しかし，公平に見て，この大胆な推測を完全に探求するために行った知的努力の程度は，アプリケーション側からの他の並列モデルと比較して，はるかに大きいと言えます。理論面を言っているのではありません。他言語のコンパイル方法，アプリケーション作成方法などを指しているのです。

この並列理論型プログラミング分野において，我々は今までにかなり完ぺきな仕事を成し遂げたと自慢しても良いと思います。そして，少なくとも，この抽象化の層を効率良くインプリメントすれば，非常に有用なものになるということに関しては，自信を持っていいと思います。

では，答えを続けましょう。

Dally：それが有用であることは間違いありません。皆のスライドを見ると，図の真ん中がくびれて，上と下が太いわけですが，我々が考慮したいと思う多数のモデルやプログラミングがありました。それらは論理型プログラミングの様々な形か，あるいは，並列データ，データフロー，アクター・プログラムを書きたい人へと拡張し，中間部分には，好みの核言語がありました。底辺へ行くと，たくさんのインプリメンテーションがありました。真の問題は，中間部分に小型の核言語を採用するための正しい基準は何か，ということです。基準は2つあると

思います。ひとつは底辺から見上げることです。これは実は，効果的にインプリメントできるものです。もう片方は，天辺からできる限り広い視野で下を見下ろすことです。何らかの形で特殊化すると，これはあまりうまくいきません。さまざまな意味で，論理型プログラミングは，最低辺抽象化レベルとして不完全なものです。その理由は，オブジェクト指向言語またはアクター言語のインプリメント，あるいは，CSPやOCCAMのようなより同期的な並列言語のインプリメントにあります。どちらの場合も，計算から離れようとしていたり，集合しようとするアクターにメッセージを送るプロセスは，それに応答する原始的な通信動作や同期動作により，ハードウェア中に簡単にインプリメントすることができます。しかし，論理型プログラミングを行わなければならないとすると，ストリームの上の方でconsプリミティブを使う破目に陥ります。この際，メモリを割り当てたり，ストリームの変数名を導入したりしなければなりません。そこで，非常に原始的演算であるはずのものの下に，はるかに多くの機構を置こうとします。なぜなら，それは直接ハードウェアにマッピングを行うからです。論理型プログラミングあるいは核となる論理型プログラミング言語は，木の上の枝一本とでもいったものの上に立派な層を作り，ここに核となる論理型プログラミング言語にマッピングする論理型プログラミング例のすべてがあると言えるかもしれません。しかし，それよりも下のどこかに，より基本的な要素セットがあり，それが論理型プログラミングだけでなく，より広範囲な抽象化セットのコンパイル目標である必要があるように思えます。

SHAPIRO：質問があります。ある実際の言語に対し，2種のインプリメンテーション方法があるとします。ひとつは直接ハードウェアへ向かい，もうひとつは，並列論理型プログラミング言語を介します。どの程度のオーバーヘッド以下ならば，論理型プログラミング言語のプログラミング環境オペレーティング・システム等々を使ったり，直接のコンパイルを行うのではなく実際の言語を使ったりすることが実用的だとお考えですか。スレショルドはどのぐらいですか。

Dally：スレショルドがわかっているという自信はありません。しかし，ハードウェアへ直接コンパイルすることはすすめません。抽象マシンという概念は，投資節約にとって非常に重要で，人々が並列アーキテクチャの性質をもっと深く理解すれば，いろいろなマシンが現われたり消えたりするでしょう。それは始まったばかりです。種々のものを動かそうとする時，アプリケーションとシステム・プログラムの両方で，プログラミングへの投資を節約できる方が良いでしょう。

SHAPIRO：それは並列理論型言語のような抽象化の可能な分野でのポイントです。質問を繰り返したいのですが。

Dally：オーバーヘッドという意味でのスレショルドですか。

SHAPIRO：そうです。たとえば，5，10，20，100倍低いとすると，どこがスレショルドで，「プログラミング環境，オペレーティング・システム，移植性等々を得るために，よろこんで並列論理型言語と中間言語を使い，その代わりに性能をX下げることにします」と言えるのですか。つまり，何が…。

Dally：つまり，真の質問は，どんな見返りがあるか，ということですね。長期的な見返りを得るとすると…。

SHAPIRO：この言語で作成可能なすべてのソフトウェア。

HEWITT：Robinと全く同じ意見ですが，もう少し論点を整理できるかもしれません。我々が必要としているのは，ある特定のプログ

ラミング言語ではなく，これらコンカレントシステムを構築するための基盤とモデルだと思います。なぜなら，関数型であろうが，何であろうが，あらゆるタイプのプログラミング言語をサポートしたいからです。では，層についてはどう考えるのか。Roginが指摘したように，それは基本的に通信を土台とするものとなるはずです。そして，ハードウェアとソフトウェアの間の層を決定するのは，基本的には，通信という考え方と，コンカレント・オブジェクトであれ，アクターであれ，何らかの個体の概念です。

そして，アクターというアプローチをとらないこれら言語の多くは，我々の基盤の一部とすべきでない余計な荷物をかかえることになりそうです。そして，これらGHC言語における余計な荷物ということで言えば，それには2種類あると思います。つまり，同一化とガードです。それらは歴史的理由等々により，入ってきたもので，我々が創造しようとしているこのようなタイプの層における通信やアフターという考え方にとり，真に根本的なものとは言えません。

SHAPIRO：上田さん，これに答えますか？…それでは，私が答えましょう。

同一化は非常にパワフルな演算です。そして，前に私が言おうとしたポイントは，それをフルに使わなければ，car,cdr,またはconsのコスト以外には，コストはかからないということです。ガードに関しては，ガードはCCS,CSP,OCCAMにもあり，あらゆる並列論理型言語が，何らかの形でカードを備えています。

HEWITT：GHC言語にはガードがあります。あらゆる並列型プログラミング言語にそれがあるわけではありません。

SHAPIRO：CCSやCSPでは，ガードは非決定的選択の前に，何らかの方法で隠されています。ただ，アクターは例外でしょう。

HEWITT：Bill Dallyの言語やその他いろいろありますが，ガードを持たないオブジェクト指向並列言語は，非常にたくさんあります。

SHAPIRO：近山さん，どうぞ。

近山：小さな構造を使った通信について一言。実際にconsセルを割りあてずに2つのプロセッサ間で通信できるような最適化があります。ただし，言語レベルでは，consセルを使います。このような最適化は，すでに存在しています。私が言いたかったことのほとんどはあなたが言ってくれたので，コメントだけにします。つまり，一種の最適化も可能だということです。consセルをうまく割りあてることにより，オーバーヘッドを除去することさえできます。

HEWITT：余計な荷物について語る場合，それには2つの部分があります。ひとつは，余計な荷物が無いかのように速くマシンを走らせようとする場合，どのような最適化を行うかということです。それは，特殊なケースにおける最適化ということで以前に触れたような技術を用い，何とか処理できる部分です。余計な荷物の残りの部分は，常にこういったことを考えていなければならないという精神面での余計な荷物です。ほとんどの場合は特殊ケースしか使わないとしても，やはりこれらすべてのガードとこの複雑な単一化に対する説明を，心のどこかに留めておかねばなりません。

SHAPIRO：Leon。

STERLING：話題を少し元に戻して，第五世代プロジェクトの中心テーマについて話したいと思います。並列性に関する考え方が，人工知能の今後に意味するところについて，何かコメントしてもらえますか。

SHAPIRO：それはパネリストの皆さんの専門外の分野ですが，どなたか意見があれば…。

HEWITT：私がやってみましょう。この点に関しては，従来の概念を超えるという意味で，難関が横たわっていると言えます。なぜなら，これら大規模なコンカレント・マシンでは，人工知能で従来使われている方法でそれらを取り

扱うことも，プログラムすることもできないからです。どうにかして，大規模組織，人間のグループ，そういったものを取り扱う技術を持ち込む必要があると思います。この特定のルートだけが，我々が進みつづけることができる唯一の道であると思います。なぜなら，当初Turingから得た人工知能に関する古い概念は，何か知的なものを作りたいという願いから発したきわめて心理学的な概念であったからです。多くのアクターと行ったり来たりする何十億ものメッセージにより，我々はオープン・システムで稼働する新しいタイプの組織をサポートすることができます。

WARREN：並列推論マシンの構築は，人工知能の問題を直接解決するものではありません。全く解決できません。しかし，問題と取り組むためのよりパワフルなツールを提供してくれます。

MILNER：私はDavidと同じ意見です。人工知能はその性質から言って，もっとも困難な問題ですから，我々はツールの理解に努めるべきでしょう。そして，ツールは将来，大部分並列になるでしょう。

（フロマ）：もう少し焦点を絞りましょう。

すべての発表に関して共通と思われたことは，世界を完全な離散事象上での個々の客体に分解しているという点です。また，言語学的抽象化という考え方も共通していると思います。この点について，皆さんの意見を聞いてもいいでしょうか。それが世界を把握する場合の基本的かつ正しい方法だと思いますか。Carl Hewittさんの考えをもう少し押し進めれば，個々のトランザクションの結果を知らない場合，そういった世界の見方には，潜在的問題があるということが言えるのではないかと思います。これらのマシンの窮極的な見込みや，知的か知的でないかについての意見を述べる場合に，これらのマシンを個々の客体の考えをベースとして，設計し，構築することを前提としているのです。

Dally：世界は何らかのレベルで先天的に離散したもので，コンピュータ世界はそれよりもさらに離散しています。なぜなら，ディジタル・コンピュータの構築に制限するとすると，基本的に，ビットを記憶する記憶セルを持っています。我々がコンピュータ・システムにできることと言えば，それら記憶セルの接続方法とその上に定義する状態遷移関数の指定ぐらいです。それは，有限の状態変化を扱うコンピュータ設計の非常に基本的なレベルです。これまでに見たモデルすべてがそのレベルから構築しなければならないので，それらはある意味でコミュニケートし合っています。言語学的抽象化の質問については，誰でも一度にマシン中のあらゆることを考えたくないからです。正しいモデルと正しい抽象化を用意すれば，それらを利用して，何かの役に立てることができます。

Milner：それに付け加えたいことがあります。確かに，並列プログラムとして表現可能な形に世界を分けることはできそうですが，その別々の部分は，必らずしもシステムの物理部分に対応していません。フランスのSophia Antipolisで，Gerard Berryがたいへんおもしろい研究をしています。彼らは一種のプロセス代数学でリアルタイム並列システムのモデリングを行っているのですが，通常は通信と考えるような個々の事象間の同期化が，実はひとつのシステムの仮想部分あるいは異なる見方であることがわかったのです。つまり，それは，各コンポーネントが現実のマシンの物理部分ではなく，マシンの状態構造の部分のひとつの見方であるような有限状態オートマトンの分解を発見するようなものです。そのようなものの合成例として，彼は腕時計等のモデルも提示しました。Harelの状態図の研究も，これと似ていると思います。したがって，プロセス代数学や事象にもとづくモデルの世界では，部分を物理部分と見做すこ

ともできますが，必ずしもそう見做さなくても良いのです。

Snapiro：上田さん。

上田：人工知能のようなアプリケーションに関していえば，このプロジェクトは，目標を非常に遠くに置いています。10年前，我々はアプリケーションとハードウェア間のセマンティング・ギャップは，どんどん狭くなると期待していました。しかし，現在，それは広がるばかりのように見えます。

アプリケーションは複雑度を増しています。が，一方，コンピュータ・アーキテクチャは単純化してきています。ハードウェアのレベルでは，現在，オブジェクトの物理分散を考えなければなりません。この事実に直面し，我々がすべきことは，これら両端を結つける適切な概念を多数見つけることですが，これは容易ではありません。我々が核言語に関する研究によって，確立しょうとしているのは，このプロジェクトの両極を結びつける役に立つ概念のひとつです。

Shapiro：古川さん。

古川：まず，我々の所期の目的は，非常にすぐれた人工知能を創造することではなく，知識情報処理システムの研究に対する基礎を作ることでした。コンカンシーとパラレリズムは，いくつかの面で密接に関係しています。ひとつはもちろん，処理能力の増大であり，もうひとつは，知的能力の向上です。オブジェクトの並列性も，この範囲で扱わねばなりません。もうひとつはアプローチとして，我々はハードウェアとアプリケーションの橋渡しをするたくさんの層を準備中です。また，容易とは思いませんが，部分評価によるアプローチにも賛成です。今後必要なことは，高級記述から低級記述への変換を可能にする多種多様な手段を開発することだと思います。

DALLY：上田さんのコメントにさらに付け加えると，ハードウェアとアプリケーション間のセマンティックスのギャップが広がりつつあるのなら，それは2つの面からより広く理解できるようになったことを反映しているのだから，むしろ良いことではないでしょうか。

まず，ハードウェアを下位レベルに下ろすということは，ハードウェアに何を盛り込めば良いかということに関する理解が深まったということです。実際には必要のない多くの複雑な機能をハードウェアに背負わせていた時代もありました。それらを取り除くことで，はるかに効率の良いハードウェアを構築しました。同時に，プログラミングのレベルを引き上げ，プログラミングがハードウェアの近くにあった時よりもはるかにパワフルなアプリケーションを作成することができます。したがって，セマンティックスでの大きなギャップは，狭くしようとするのではなく，むしろ求めるべきではないでしょうか。

WOOD：オーストラリアのJames Woodです。国に帰ったら，たった今私が論文で読んだPrologのことを聞いた人たちから，64,000個のプロセッサを備えたコネクションマシンについて聞かれると思います。このPIMまたは64個のプロセッサを持つマルチPSIについてはどうでしょうか。そんなに注目されているのですか。パネリストのどなたかで，1文か1段落で答えてくださる方はいませんか。答えがもらえれば，国に持ち帰ることができて，また海外へ出してもらえるのですが。

DALLY：プロセッサという単位はどんなものに関しても，尺度としては役に立ちません。どうですか。これなら1文ですよ。

つまり，あるものからある数を引き出し，よく似ているように見える数の比較を始める，ということをよくやります。ここではそれは通用しません。Waltz博士ならば私よりも多くを語れるはずですが，コネクションマシンはたくさんの1ビット・プロセッサを持っています。それらは2つの8対1のマルチプレキサと考える

ことができます。実際，まさにそれなのです。それらが3ビットを取り込み，任意の2のテーブルを入れると，2ビットを生成します。それは構築を始めるための非常に小さい要素です。ですから，それとPSI中のプロセッサを比較することは難しいのです。後者はパワフルな40ビット・マシンで，論理やリストを扱う言語を処理するためのいろいろなメカニズムを内蔵しているからです。

David Waltz(Thinking Machine社，U.S.A)：汎用性とコンカレンシーの面の目的の両方について質問があります。

コンカレンシーのもっとも面白い源泉は，実世界，たとえば実際のビジュアル世界のように思えます。人間が行うことの多くは，さまざまな事象のコンカレンシーを理解することです。我々は偶発性やさまざまな領域が互いに結びつけられることをこのように理解しています。これが，真実なら，優れた並列言語であれば，視覚などを扱うこともできるはずです。さて，私の見落しでなければ，まだどなたも論理型プログラミングやその派生物と視覚とを関連づけるような発言をしていないようですが…。

DALLY：ほとんどの部分で，論理型プログラミングとその派生物，親類は，視覚処理のようなものとはほとんど関係ないように思えます。一方，コネクションマシンのような他種マシンにおいては，それらは自然な問題で，論理の抽象化や私がラフな推論と呼ぶものとは，比較的関連性が薄いと思います。ある意味で，あなたが必要としているものは両者を含む包括的理論であり，それはその両者の個々の見方からは，引き出し得ないものかもしれません。推論だけでなく視覚に関する作業にも充分な優れた包括的な並列理論が，どんなものかについてのコメントになっていると良いのですが。

SHAPIRO：質問に答えたいと思います。たぶん他の人もそうでしょう。

視覚および論理型プログラミングにおける単純できわめて一定の処理コネクションと論理プログラミング間の関係を考えるひとつの方法は，コネクションマシンにインプリメントしようとするアルゴリズムの仕様を与えるシストリック的プログラムを並列論理型プログラムで作成することは簡単だということです。つまり，これらのアルゴリズムの形式化と仕様記述には，並列理論型言語はきわめて良いツールであり，それを示す良い例がいろいろあります。これらの言語のコネクションマシン上でのインプリメントに対してさえ今日利用できる技術を考えると，おそらくこれは長い道のりでしょう。したがって，答えは，現在は精神上または形式上の表現を部分的に評価し，人手によってでC*のコネクションマシン・プログラムを作成するということになるでしょう。しかし，いつか将来，これらの高級記述をコネクションマシンのような高度に特殊化したマシンにマッピングするためのコンパイラか部分評価機能が現れるかもしれません。つまり，私が提示した見方では，コネクションマシンのような特殊化したマシン上での高度並列アルゴリズムを否定してはいません。ただし，ソフトウェア技術とそれの自動化は，まだまだ実現からはほど遠いということです。

WARREN：視覚は論理型プログラミングという観点から，面白い問題だと思います。特に，高性能の並列システムに関して，それが言えます。視覚は，もちろん，上のレベルから下のレベルまで，非常に広範囲です。しかし，たとえば，群衆の中から一人の顔を探すというような場合，AND並列とOR並列を持つシステムを考えてみましょう。OR並列は，場面全体をサーチして可能性のある顔を見つけようとし，AND並列はその位置にある特定の顔を分析し，それが探している顔であるかどうかを判断しようとします。したがって，これは考えてみる価値のあるアプリケーションであると思います。

MILNER：数学的モデルについて簡単に触れてもいいですか。視覚のように精巧な，特殊化したものを考えることになると，コンピューティングにおける基本的モデルのいくつかが全く役に立たないという事実は，とても面白いと思います。そこで，我々は，コンカレンシーの基本概念は扱っているものの，構造の一定性を利用する能力を持たない一般計算モデルと，ある特定の問題によく適応したマシン構造との間の大きなコントラストに驚くわけです。そういったモデルと特殊モデルの間のギャップは非常に大きなものです。モデル構築を専門とする者として，視覚のような問題に直面すると，大きな屈辱を覚えます。おそらく汎用プログラミング言語でもそれらを表現できるでしょうが，特殊プログラミング言語による表現の方がはるかに容易であることが判明するかもしれません。視覚には，特殊プログラミング手法あるいは特殊プログラミング言語を利用すると良いのかもしれません。

DALLY：モデル構築ではなくマシン構築に携わっている私としては，視覚の問題で屈辱感を味わうことはありません。たとえば，コネクションマシンが視覚の取扱いに対して行うことを見ると，１ビット値を簡単に扱うためのデータ型の特殊化ということです。しかし，初期視覚に対してさえも，人は６から12ビットの範囲（これはカメラ・システムの解像度によって変わってきますが）のものを扱うことの方に関心を向けます。したがって，１ビット値のテンタ型の構造化を行う余地があります。もうひとつは，マシンの局所性を利用できるような通信の特殊化です。適切な要素メカニズムを構築できれば，汎用マシンからそれらの両方を得ることが可能です。小さな整数に対するデータ型演算を，きわめて効率良くインプリメントし，また，現在入手可能な最高速の結線を用いて，隣接のプロセッサと高効率で通信することができます。そして，計算の非同期モデルやグローバル共用メモリを必要とするモデルをサポートするための一般性を犠牲にせずに，これらすべてを行えます。

これは論理型プログラミングでは難しいかもしれません。ただ，視覚をうまくサポートできるマシンがあるとは思います。このようなマシンは，いくつかの視覚機能については，ハードワイヤされた専用視覚マシンとは競争できないでしょう。しかし，プロトタイプ作成では，その代わりをつとめることもできるでしょう。

Shapiro：Waltzさん。討論を続けたいですか？

Waltz：はい。これまで聞いた答えのどれにも，何らかの真理があるようです。何と表現したらよいかわかりませんが，皆さんがもっとも関心を抱いている言語ファミリーにおける研究の基本的努力は，皆さんが注目すべきもの，および特に組み合わせるべきものを選び出すことに集中しているように思えます。リソースの効率良い利用は，それらが適用されるべき部分にのみ，適用されています。

真の知性は，種々のプロセッサが働いているが，その大部分は，ほとんどすべての時間にわたって無関係な仕事しかしていないようなプロセッサを浪費する処理を必要としていると思います。言いかえると，ほんとうに役に立つことをしていないプロセッサがたくさんあります。少数のプロセッサを使えば，効率の良い構築が行えますが，応答時間を速くするために多数のプロセッサを使うということだと思います。

上田：ちょっと一言。これまでは並列計算をプログラムすることを目的として並列論理型言語を使ってきましたが，これからは，記憶をプログラムするのにも並列論理型言語を使い，応用範囲をデータベースその他へ拡げてゆこうと計画しています。このように原始的言語が，広範囲なアプリケーションをカバーできるのです。

SHAPIRO：ある意味で，並列論理型言語は低級アルゴリズム言語であり，どのような形式のアルゴリズムをインプリメントしたいかという点に関しては，どのような立場も取っていません。少くとも，そこが我々が研究しようとしているポイントです。他にコメントはありませんか。どうぞ名前をおっしゃって下さい。

伊藤貴康：東北大学の伊藤貴康です。コンカレンシーの現象においてもっとも重要で難しい概念は，デッドロックの存在だと思います。したがって，デッドロックの存在の証明とデットロックからの回復は，コンカレント・システムの理論と実際にとってたいへん重要です。しかし，この討論では，デッドロックに関して何も話されていません。皆さん，デッドロックの心配はないのでしょうか。

MILNER：CCSで構築したソフトウェアがありますが，それが最近，VAX VMSメイリング・システムで，デッドロックを発見しました。それは非常に短い活動で，修復作業中です。つまり，特定の理論にもとづき，デッドロックの発見と修復を助けるソフトウェアを構築できるということです。それはきわめて容易なことだということだけは言えます。等価性や特定の特性を証明することは非常に困難ですが，通信システムの解析でデッドロックを発見することは，きわめて容易です。

HEWITT：デッドロックに関しては，基本的に，どこへも動かないアクターの中にトランザクションが居すわっているという点にまで，コストを下げることができるでしょう。最終的には，タイムアウトを介して，クリーンアップすることができます。しかし，コンカレント・システムをくっつけるやり方を与える組織構造をより細かくすれば，デッドロックは完全に無くなるはずです。永久に完了しないトランザクションはありませんが，最終的にはそれらはシステムから掃き出されてしまいます。

伊藤：部分評価の重要性が指摘されていますが，部分評価をするとデッドロックの解消やデッドロックの存在証明に新たな困難な問題が生じると思いますが，どうでしょうか？

SHAPIRO：まず，デッドロックの可能性は，あらゆる並列形式や並列プログラミング言語が先天的に持っているものです。ある意味で，デッドロックを示せない言語は並列プログラミング言語とは呼べないと言ってもいいくらいです。

デッドロックの検出あるいはその不在の証明に関しては，Meta88会議におけるJohn Gallagher，Codishと私の論文があります。そこでは，抽象解釈の技術を用い，並列論理型プログラムにおけるデッドロックの行動を分析する方法を示しました。つまり，それは部分評価ではなく，抽象解釈です。ただし，これらは密接に関係し合う技術ですが。また，CCS技術にも関係があるはずです。

さて，討論も終わりに近づいたようです。質問をあと1つだけ受けつけましょう。その後，パネリストの皆さんから締めくくりの言葉をいただきます。

質問：AIおよび知識表現の観点から話したいと思います。

Milner教授にうかがいたいのですが，歴史的に言って，現在の論理型プログラミングに関する研究は，論理はたいへん優れた知識表現方法であり，人工知能が必要とするほとんどの知識を表現できるという仮定のもとに発展してきました。これより，Prologと論理型プログラミングが生まれました。

教授の言う新しいタイプの数学で直接コンカレンシーに攻撃をかけた場合，この新しいタイプの数学が，このネットワークや論理やその他いろいろな方法とは異なる知識表現方法を生み出せると思いますか。

MILNER：そう願っています。その可能性

は残しておきたいと思います。また，様相論理は，コンカレンシーが関係する事象システムに非常に適しているということも述べておきたいと思います。2つの敵対し合う党派を結びつける新しい手段として，様相論理を考えても良いかもしれません。つまり，様相論理は，論理型プログラミングの表明面での長所を持ち，一方，プロセス代数学にも非常に近いのです。

あなたの質問に対する完全な答えとはならないかもしれませんが，可能性はあると思います。解決に向かって蓄積は進んでいると思います。

座長：パネリストの皆さんに，この討論会の締めくくりとして，まとめの言葉をいただき，この討論を終わらせたいと思います。

DALLY：何らかのレベルで一種の抽象マシンがあるという点には，誰もが同意しているようですが，その点から，アーキテキクチャに関して一言述べたいと思います。アーキテクトにとっての課題は，今日利用できるものを最善の方法でワイヤすることでも，好みの言語学上のモデルをハードワイヤすることでもなく，何らかの一般的な効率の良いメカニズムを発見することです。そして，PIMやマルチPSIのプロセッサがコネクションマシンのプロセッサと同等でないということがありましたがマシンにおけるコストについて非常に注意深く考える必要があります。それは，ある特定の時点で何台のプロセッサあるいは何％のプロセッサを使っているかということではなく，マシン構築のコストと問題解決の速度の兼ね合いで考えるべきです。

メカニズムについてもう一点言えることは，アーキテクチャの観点から抽象マシン・モデルを構築するための適切なメカニズムは，言語学上あるいは記述上の観点から選択するものとは一致しないだろうということです。

HEWITT：今は大変エキサイティングな時代で，特に，新世代コンカレント・システムのことを思うと，その感を強くします。ICOTは当初からコンカレンシーを絶対的根本問題であると正しく判断してそれに集中し，同時に，それを高次情報システムと結びつけました。これは，事の核心において，まったく正しかったと言えます。実際，これは，ビジネスを行う場合の意識的思考や証明方法などの逐次的な処理をモデルとする心理学的考え方にもとづく人工知能から，コミックメント，ネゴシエーション，責任，等々にもとづく社会学的アプローチへの移行を示しています。それは逐次ではなく，根本的に並列であり，議論と証明にもとづくのではなく，根本的にコミットメントとネゴシエーションを土台としています。

NILNER：私は大胆な仮説を信じており，その仮説がよりうまく描写できれば良いと思っています。互いに反発し合う仮説を出しあいましょう。また，推論の方が動作よりも基本的だと説得されてはいない，ということも述べておきたいと思います。いまだに私は動作の方が推測よりも基本的ではないだろうかと考えています。

上田：該言語研究の二つの方向性を提示してたと思います。現在の核言語には，2つの問題があります。ひとつは，システム・プログラミングに対するエレガントな枠組みができていないという点で，弱いということです。したがって，並列計算のリフレクション機能を考えていかなければなりません。

二番目の問題は，基本メカニズムである単一化を，並列マシンに効率良くインプリメントすることが難かしいという点で，現在の核言語は強すぎるということです。我々は，並列プログラミングを促進するため，核言語の非常に効率の良いサブセットの設計を計画しています。

WARREN：私の最近のポイントである「コンカレンシー」と「パラレリズム」の区別に戻りたいと思います。アプリケーションにつ

いて考えると，多くのアプリケーションで，コンカレンシーは意味のある問題ではありますが，だいたいにおいて，我々が解決しようとしている問題の中心テーマとは言えません。その観点から見れば，今日討論してきたことは，解決すべき問題全体のごく一部にすぎません。

パラレリズムについては，計算の高速化という点は，小さいけれども重要な領域だと思います。パラレリズムは，できる限り底辺に置き，普通のユーザには見えないようにすべきです。

座長：この会議同様，このパネル討論会でも，この研究分野の成熟度と，そして，異なるアプローチを取る人々の間で，同意は得られないにしても，少くとも，知的に話し合うことができるということが示されました。これは，知的方法で議論できる程度まで，我々が互いの解釈と考え方を理解し合い，互いに実りあるアイディアを交換できるということを表わしていると思います。

パネリストの皆さんと会場の皆さんのご協力に感謝します。ありがとうございました。