たしている

論理が計算機科学の中心であると信ずる理由を挙げてきました。また，論理は人工知能の重要な要素ではあるが，それが人工知能の中心をなすわけではないと考える理由についても述べました。時がたてば，きっと論理は今以上に計算機科学において重要な役割を占めているでしょう。人工知能では，論理は，心理学，神経科学，言語学など関連する他の多くの学問の中の1つにすぎません。計算機科学ならびに計算機の歴史に登場する傑出した人物は，TuringとvonNeumannという2人だけであると述べました。この2人の研究者は，どちらも最初から計算機分野で主導的な立場にあったが，この2人の中で科学的に重要な役割を果たしたのはTuringの方です。Turingという名前は，計算機科学全体を言い表す唯一の名前です。人工知能の研究に関しては，Turingが，他の研究者をはるかに引き離して独走していたことは間違いありません。

したがって，1992年6月23日は，Turing生誕80周年記念を思い起こすのにふさわしい日です。第五世代コンピュータシステムプロジェクトの大フィナーレである，このすばらしい会議とそれが我々に示すものは，Turingの生誕記念に最もふさわしいだけでなく，Turingの注目すべき科学的貢献と彼が創り出しかつ第1人者として発表した概念を記念するものであると思います。Turingが生きていれば，きっとこの機会を暖かく見守ってくれただろうと思います。

ありがとうございました。

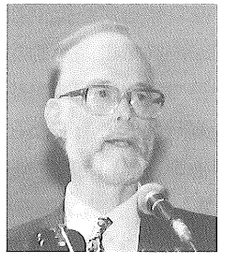
質問：あなたの講演はすばらしかったと思います。計算機科学における論理の役割を理解することに関して，1つ意見があります。我々は，おそらく論理の役割と，対照的な哲学の合理論と経験論の役割を比較しているのかもしれないということです。論理は，計算機科学という学問の合理的な側面です。論理プログラミングは，論理推論という点ですべての物事を合理的に表現しようとします。しかし，実際の世界は経験に基づくので，実際の世界がどのように構成されているか理解するには，論理以外の経験に基づく手法を使用しなければならないかもしれません。たとえば，計算機科学では，オブジェクト指向プログラミングは世界のモデル化に対して経験的なアプローチを採用していますが，論理プログラミングは合理的アプローチを採用しています。アプリケーションを行うときには，計算機科学と人工知能の分野で論理を超越する必要があると思います。

回答：はい。その意見に賛成です。それは，基本的にMinskyが人間の動作を理解するというケースで我々に教えてくれていることであると思いますが，もっと一般的にすべての自然現象や自然の過程にも全面的に適用することができます。

自然界で見られる事象は，おそらくあまり論理は関係していないでしょう。しかし，ここで強調しておきたいこと，そしてあなたの類推には出てこないことは，論理は人工的な産物(自然界では発生しないものであると定義する)において，一段と重要な役割を果たすかもしれないということです。我々自身は，人工的に物を設計構築しますが，そうすることによって自然が自ら生み出してきた概念や技術に限定されるわけではありません。Herb Simonの言葉を借りれば，人工科学において，論理はもっと中心的な役割を果たすことが可能であるし，実際に現在その通りであるし，これまでもその通りであったし，また将来もその通りであるということです。これは，私が論文の中で強調したい点です。計算機は自然界には存在せず，我々が創り出した物です。それによって，論理を主な指針となる概念として使用し，いわばそれを合理的にすることが可能なのです。

招待講演

FGCS'92



述語としてのプログラム

オックスフォード大学教授(英国)
C. A. R. Hoare

第五世代コンピュータシステムプロジェクトの終了を記念する，この会議において講演をさせていただくのは誠に光栄なことです。皆様の偉大な前進と業績に対して，皆様を賞賛あるいは支持する多くの人々とともに，祝辞を述べさせていただきたいと思います。

このプロジェクトは，コンピュータ技術の大幅な進歩を目指すだけでなく，その技術を人類の役に立つものにするという，野心的かつ気高い目標を掲げてスタートしました。困難な課題は数多く残っていますが，この目標に向かって，長い間科学者と技術者が最善の努力を続けてきました。

革新的な新しい概念の創出とその成熟のためには，10年間にわたる資金提供が必要なことを認識していた日本の通産省の先見の明を讃えたいと思います。また，このプロジェクトを率い，また取り組んだ多くの方々，その多くは今日ここにいらっしゃいますが，その方々の勇気と献身的な活動を讃えたいと思います。皆様は，計算機科学の分野で理論と実践の両方の発展に貢献してきたわけです。さらに重要なことに皆様は，このテーマの理論と実践との結びつきを発見し，利用してきました。

当初から皆様は，実用的な計算機科学の最も重要な問題の中の，2つのテーマに努力を集中してきました。そのテーマとは，高並列処理技術の効率的な利用法と，高度に記号的なデータ処理の先進的アプリケーションです。皆様は，論理プログラミングとProlog言語の論理的特性からインスピレーションを得ました。論理的アプローチだけが，新世代コンピュータとそのユーザ向けのプログラムを作成するという厳しい仕事を解決できるという見方をとりました。

私も同じ考え方に刺激を受けてきました。プログラミングの仕事は常にユーザの要求と目的の明確かつ単純な表明で始まるべきであり，その表明は，プログラムが果たすべき目的の仕様として形式化することができるという考え方を，私も以前からとっています。そのような仕様とは，得ようとしているプログラムの動作の観測結果として許容されるすべてのものを記述した述語なのです。

述語

述語の自由変数は，実行中のプログラムの動作—あらゆる状況での動作—に関して観測しうる内容を表します。述語は，プログラムの実行時にこれらの変数がとることのできる，すべての許容値を記述します。仕様に関して最も重要な要件は明確さであり，そのためには可能な限り高い抽象度，モジュール性および表現力をもった記法が必要です。仕様が，プログラムがどう動いてほしいかを明白に記述していないとすると，望まざるものを記述している危険が高くなります。このような危険性をチェックすることは，困難で費用が掛かり，

また数学的に不可能です。

仕様記述言語に関する最小限の要件は，ブール論理の基本的な結合子を完全に一般的な形で包含することによって，仕様を連言，選言，否定によって，つまり単純な“AND”，“OR”，“NOT”によって結合できるようになっているべきであるということです。連言は，たとえば，プログラムで圧力『および』温度を制御する場合のように，満足させなければならない2つの要求を結合するのに必要です。選言は，システムの実現において誤差を許容するのに必要です。システムは，最適温度から1『または』2度ずれてもよいかもしれません。否定は，「爆発してはなら『ない』」というような要求を書くという，さらに重要な理由で必要です。

通常の論理の表現力から，「Pであるか，またはPでない」といった仕様を書くことは可能です。これはもちろん，常に真であって，考えられる限りのプログラムのあらゆる観測内容を記述しています。そのような寛容な仕様は，簡単に実現できます。無限ループに入ってしまうプログラムをもってきてもいいのです。

もう1つの極端なケースとして「PであってPでない」という仕様があります。これは当然のことながら常に偽です。そのような矛盾する仕様を満足する製品はありません。このような仕様は，インプリメンテーションを始める前に除去すべきエラーの兆候を示すものです。

論理プログラミング言語の設計者と共有している，もう1つの興味深い洞察は，プログラムもまた述語であるという点です。適切に解釈すれば，プログラムは，実行中のプログラムの動作の観測結果としてありうるすべてのものを記述しています。プログラミング言語は，単純な構成要素から複雑なプログラムを組み上げるための結合子をいろいろ提供します。たとえば，Prologは，カンマ(,)と表記し“and then”と読む一種の逐次合成，セミコロン(;)と記し，“or then”と読む一種の逐次選言，およびチルド(~)と表記し，おそらく“Impossible”と読む一種の強否定を提供します。以上の演算子は，コンピュータでの効率的な実行のために特に設計したもので，仕様記述に使用するブール結合子，つまり実行可能であることよりは明確かつ単純であることを意図したものとは全く異なります。

それにもかかわらず，実行可能な演算子に対しても，述語論理の中だけで，やや複雑にはなりますが正確な解釈を与えることができます。この点については，これからお話しします。この洞察は，単なるPrologよりはずっと一般的であり，他の多くの言語，それどころか何らかの意味のある設計記法で記述できるどんな工学的製品にも適用できると思います。

この見方は工学の一般原理の基礎となるものであり，これからそれをハードウェアの設計，手続き型プログラム，およびPrologプログラムの手続き的解釈に適用することによって簡単に説明します。しかし，すべての述語をプログラムとして読むことができると主張するのは，全くおかしなことです。前述の「PであってPでない」という，矛盾する常に偽の述語を考えてみてください。この矛盾する性質をもった答を出してくれるコンピュータプログラムなどありえません。そこで，この述語はプログラムではなく，それを実行可能なプログラムへと変換できるプロセッサは存在しません。実現可能なプログラムに対して実現不能とわかっている動作を結びつける理論を作っても，それは誤った理論であり，使うのは危険であるに違いありません。

それとは対照的に，当然ながら，もし“P”が完全にProlog記法で表現したプログラムであれば，(P，~P)というプログラムは，有限時間内に，または無限時間かかることによって，必ず失敗するとはいえ，完全に意味のあるプログラムです。

私が紹介している，Prologプログラムの述語としての解釈は，まさしくこの失敗するプログラムの動作を記述するものです。

したがって，プログラミング言語は，述語論理の部分集合とのみ同一視することができます。この部分集合の中の各述語は，そのプログラミング言語で表現可能なプログラムのあらゆる可能な動作を正確に記述したものです。この部分集合は，矛盾や，他のすべての実現不可能な述語を排除するように設計されていて，プログラミング言語の表記法は，この排除がきちんと守られるように注意深く設計されます。

原則的に，その表現力上の制約から，プログラミング言語は，適度に高水準の抽象度で要求仕様をモジュール形式で記述するための記法としてはあまり適していません。

プログラムと仕様はどちらも述語なので，プログラムの正当性は単純な含意によって確立することができます。“S”という仕様が与えられた場合，プログラマの課題は，その仕様を満足するプログラム“P”を見つけることです。ここで仕様Sを満足するとは，Sが，プログラムPのありとあらゆる実行に対するありとあらゆる観測結果を記述しており，したがってそれらの観測結果を許容しているという意味です。論理ではこのことを，「PならばSである」という単純な含意の証明によって，数学的な確実さで保証することができます。「プログラムや製品がその仕様に適合する」というのがどういうことかを，このように単純に説明できるというのが，プログラムと仕様の両方を一様な述語論理の中で解釈する主な理由の1つです。

これで，矛盾する述語「偽」を，プログラミング記法から排除する必要性について説明することができます。「偽」がいかなる仕様Sをも含意するというのは，初等論理の定理です。したがって，「偽」は，あらゆる仕様を満足するという驚くべき性質を持っているのです。あなたがしたいことは，何でもそのプログラムがやってくれるというわけです。幸いなことに，そのような奇跡は存在しません。というのは，もしそんな奇跡が存在するならば，その他のものは何も必要なくなるからです。まちがいなくプログラムも，プログラミング言語も，コンピュータも，そしてコンピュータ科学者やプログラマでさえも必要なくなってしまいます。

この考え方の非常に単純な例を，従来の手続き型プログラミングの領域に見い出すことができます。ここで，最も重要な観測可能値は，プログラムの起動前のマシンの状態から観測できる値と，プログラムの終了時の状態から観測できる値です。

Xという名前で，もしかするとXと呼ばれているある変数の初期値を表すことにします。X'を同じ整数変数，すなわちここで我々が注目する唯一の変数の最終的な値とします。

Procedural Program

Let x be initial value

Let x' be final value

Let $S = (x' > x)$

Let $P = (x := x + 1)$

$= (x' = x + 1)$

Conclusion:

$\vdash P \Rightarrow S$

Fig. 1

仕様Sが，「X'はXよりも大きい」という内容だとします。言い換えると，変数Xの値を増やさ

なければならないということです。プログラムPは，Xに1を足すものと定義します。このプログラムを述語として解釈すると，「最終値X'は初期値Xに1を足した値である」ということになります。(Fig. 1)

任意の初期状態で(つまりXの初期値にかかわらず)Pの動作を観測してみると，必ずこの述語を満足します。したがって「PならばSである」という含意の妥当性によって，Pが，Xの値を厳密に増やすという仕様に正しく適合していることが保証されます。

ハードウェア設計

私の考え方の一般性を示すために，次の例を，組合わせハードウェア回路の領域から引いてみましょう。これらの回路もまた，述語として解釈できます。AとBの2つの入力線と，Xという単一の出力線をもつごくふつうのANDゲートは，「XはAとBのどちらか小さい方に等しい」という単純な等式によって記述されます。これらの自由変数の値は，演算の特定のサイクルの終わりにおける同名の線の電圧として観測されるものと解します。そのとき，出力線Xの電圧は，たしかに入力線AとBの電圧のどちらか小さい方になります。(Fig. 2)

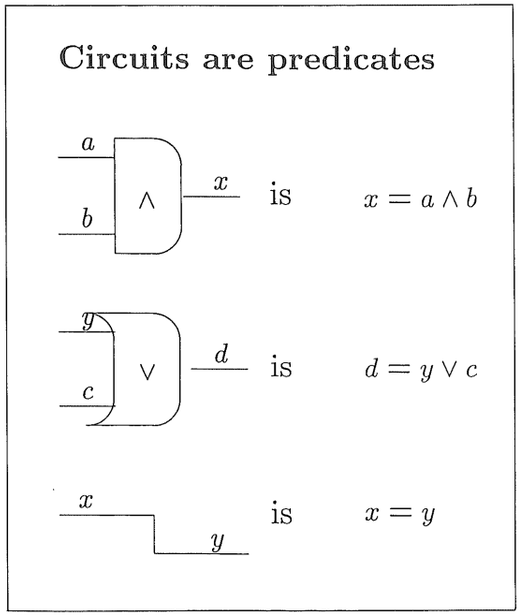


Fig. 2

同様に，ORゲートは，「Dは，YとCの大きい方に等しい」という，異なる線名をもった異なる述語によって記述できます。単なる配線は，その両端の電圧を等しく保つ素子です。たとえば，この場合は，「XはYに等しい」というようになります。

さて，並列に動作する2つの部品の組合せを考えてみましょう。たとえば，ANDゲートとORゲートを組み合わせてみます。この2つの部品を記述する2つの述語は，共通の変数を持ちません。これは，2つ部品の間に接続関係がないということを反映しています。したがって，この2要素の組合わせの動作を同時に一緒に観察しても，単に2つの動作が並行して，互いに独立して展開しているだけです。これらの動作は，個々の動作を記述した2つの述語の単なる連言によって正しく記述されます。(Fig. 3)

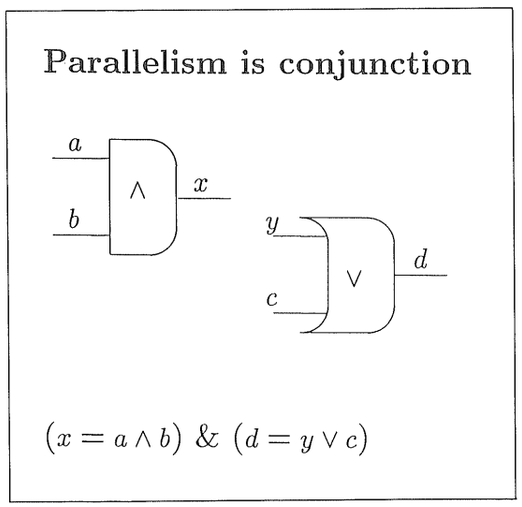


Fig. 3

この単純な例は，部品の並列的な組合せは，少なくともそれらの間に相互作用の可能性がない場合には，それらの動作を記述する述語の連言によってモデル化できるという原則を説得力を持って示しています。(Fig. 4)

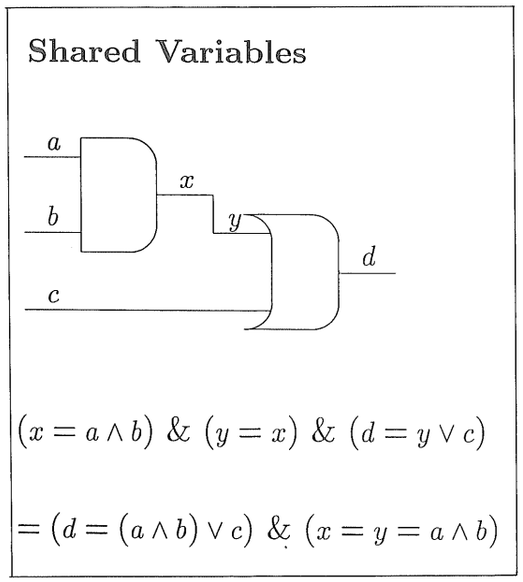


Fig. 4

部品が，それらが共有する変数で結合されているときでも，しばしばこの原則が成り立ちます。たとえば，XとYを結合する配線を回路に追加して，この表に示すような3重の連言を得ることができます。この3重の連言は，依然として組合わせたもの全体の動作を正しく記述しています。この述語を，この図の最後の行に示すように，数学的に等価な形に単純化してみました。

変数名を共有することによって，たとえばこの場合にはXとYを共有することによって構成要素を結合する場合には，その製品の利用者は，共有変数の値には全く興味や関心がないのが普通です。しかも，共有変数の値を観測する自由も，部品の組合わせをいわばブラックボックスで囲むことによってなくしてしまいます。したがって，このブラックボックスの動作を記述する述語から，それらの変数を削除しなければなりません。述語論理において，そのような自由変数を削除する標準的な方法は，限量化(quantification)によるものです。(Fig. 5)

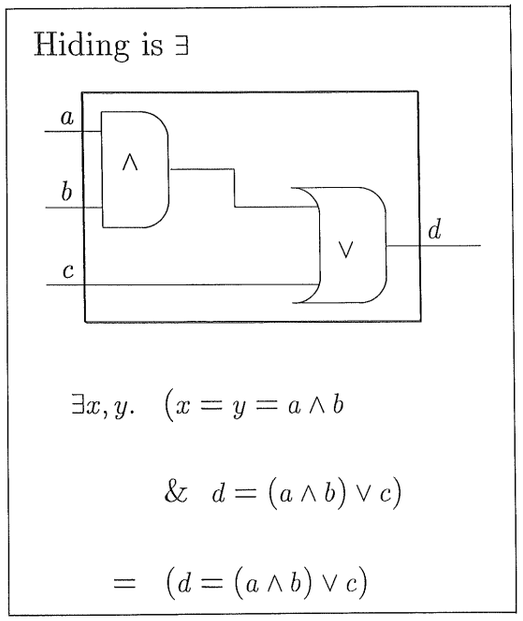


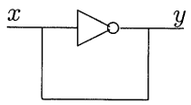
Fig. 5

工学的な設計の場合には，存在限量子による限量化が正しい選択です。XとYという隠れた変数に関しては，潜在的に観測可能な値が存在しなければなりません。しかし，その値が一体いくらになるか関心を持つ人はいません。我々のハードウェアの例では，この表に示す存在限量化された式は，「DはAとBのどちらか小さい方と，Cとの最大値に等しい」という形に簡単化できます。最後の式は，回路の外から見える配線にだけ触れていますが，おそらく我々の小さな組合せ回路が意図する機能あるいは仕様を表しています。もしそうだとしたら，皆様にお目にかけているものは，組合せ回路がその仕様を満足していることを示す単純な証明にほかなりません。

残念ながら，述語のすべての連言が，この理論において実現可能な設計にたどり着くとは限りま

せん。たとえば，「XはYの否定に等しい」という否定回路と，その回路の出力を入力に再結合する「YはXに等しい」という配線との連言を考えてみましょう。実際問題として，この組合わせによって，全く役に立たない電気的なショートや無限の振動のようなものが発生します。実際は，この回路は無益であるばかりか有害です。なぜならば，近辺の他の回路の正しい動作を妨害するでしょうし，あなたのトランジスタラジオまで干渉する可能性があるからです。(Fig. 6)

Shortcircuit is a problem



$$\text{is} \begin{cases} y = \neg x \\ \& \ x = y \end{cases}$$

$= \mathbf{False}$

which satisfies **all** specs

$\vdash$ false $\Rightarrow S$ for all S

Fig. 6

その回路が実際に満足する仕様は，「真」という無意味な仕様以外には存在しません。しかし，たった今述べた単純すぎる理論では，予測される効果は全く逆になります。この回路の動作を記述する述語は矛盾すなわち偽であり，必然的に実現不可能なものです。

この種の問題を解決する標準的な方法は，実際の組合せが陥るかもしれない，あらゆる異常動作の可能性を記述するに十分な数の新しい値と変数を理論に取り入れるというものです。回路の例では，おそらく3値論理が必要になります。高い電圧と低い電圧の他に，ボトムと呼び，$\perp$と表記する値を追加します。この値は，振動やショートが発生している配線で観測されるものとします。(Fig. 7)

Solution: three-valued logic

$\perp$ means "shortcircuit"

$\neg\perp = \perp, \quad \perp \wedge x = \perp, \quad$ etc.

$\vdash y = \neg y \quad \Rightarrow \quad y = \perp$

Moral:

Don't ignore errors.

Fig. 7

ボトム要素に対するいかなるブール演算も，ボトムという答を与えるものと定義します。これで，動作が「XはYの否定に等しい」と「YはXに等しい」の連言によって記述される，フィードバックのある回路の問題を解決できます。3値論理では，この連言はもはや偽ではありません。実際に，それはXとYの両方がショートしていることを正しく意味しています。この例の教訓は，「エラーを無視するな」ということです。述語は，あらゆる異常動作のしかたを含めて，設計したものの動作を記述しなければなりません。皆さんの設計したものがその特定のエラーに陥らないことを実際に証明または計算できるのは，設計が正しくない可能性を数学的にモデル化した理論においてだけなのです。

プログラミング言語

並列性が述語の連言だとすると，選言は，非決定性を仕様，設計，およびインプリメンテーションに導入するものであると，同じくらい単純に説明できます。PとQが述語ならば，その選言「PまたはQ」は，Pのようにふるまうかも知れないし，またはQのようにふるまうかも知れない製品を記述します。ただし，そのどちらになるかは決定しません。したがって，結果を予測したり，制御したりすることはできません。「PまたはQ」が仕様Sを満たすようにしたい場合には，PがSを満たすこと，およびQがSを満たすことの両方を証明することが必要かつ十分です。これは，まさに述語論理において選言というものを定義する原則です。選言は，述語の含意による順序づけの最小上界なのです。

プログラミング言語の最も重要な機能は再帰です。再帰(または特殊な場合の反復)があるからこそ，プログラムがその実行トレースよりも短くてすむのです。再帰的に定義されるプログラムの動作は，対応する述語の定義で再帰を使用することによって最も簡単に記述できます。

P(X)を，述語変数Xの出現を含む述語としましょう。すると，Xは，「XはPの不動点である」という等式によって再帰的に定義できます。そのような不動点の存在とその最小不動点としてのユニークな本質は，P(X)がXの単調関数の場合には，完備束と見なせるすべての述語の空間において，かの有名なTarskiの定理によって保証されています。これは，再帰の完全に非操作的な定義であり，プログラムと仕様のどちらにも等しく適用できます。

従来の逐次的なプログラミング言語では，初期状態におけるプログラム変数の値とその最終状態における値を区別することは本質的に重要です。Xがプログラムのすべての変数の初期値を観測したものを表し，X'が最終状態の値を表すことにしましょう。このどちらか一方または両方が，特殊な値ボトムを取る場合があります。この場合，ボトムは，非終了すなわち無限失敗を表しますが，それは再帰プログラムがおかしくなる最悪のケースの1つです。各プログラムは，初期状態Xと最終状態X'との関係を記述する述語です。たとえば，私が"II"と呼んでいて，ヌル演算とも呼ばれている恒等プログラムは，何もしないのですが，初期状態を何ら変更することなく成功裏に終了します。しかし，これが保証されるのは，そのプログラムが正常状態，つまりまだ失敗していない状態で起動された場合に限ります。したがって，ヌル演算プログラムは，「開始状態がボトムでなければ，最終状態が初期状態に等しい」という形の含意として定義できます。

Sequential Programming

Let x be initial state

Let x' be final state

Let $\perp$ be infinite failure

II does not change the state

$$II \stackrel{\text{def}}{=} (x \neq \perp \Rightarrow x' = x)$$

(P, Q) does P then Q

$$(P,Q)(x,x') \stackrel{\text{def}}{=} \exists y.P(x,y)\&Q(y,x')$$

Fig. 8

従来型の言語でPとQを逐次的に合成することは，Qの初期状態がPによって生じる最終状態と同じであることを意味します。しかし，PからQへと渡される中間状態の値は存在限量化によって

隠されるので，観測可能な残りの変数はPの初期状態XとQの最終状態X'だけです。2つの述語の合成の形式的な定義は，関係演算におけるふつうの関係合成の定義と同じです。(Fig. 8)

さて，逐次的合成が決して自己矛盾に陥らないようにするには，プログラミング言語の定義に注意を払わなければなりません。たとえば，Pがその最終値はゼロでないと表明し，Qがその初期値はゼロであると表明すると，それらを合成したものは即座に矛盾すなわち「偽」になってしまいます。

この場合の解決策は，プログラミング言語の制限された記法で表されるすべてのプログラムが，実は一定の「健康条件(healthiness conditions)」を必ず満足するようにすることです。逐次的プログラムの場合には，このような条件は，XまたはX'が失敗値ボトムを取るときはいつでも，そのプログラムの動作は全く予想できない，つまりどんなことでも起こりうることを言明します。(Fig. 9)

Problem

$$\Big((x' \neq 0), (x = 0)\Big) = \text{False}$$

Solution

All programs P satisfy

healthiness conditions

$$\vdash P(\bot, x'), \qquad \text{all } x'$$

$$\vdash P(x, \bot) \Rightarrow P(x, x'), \qquad \text{all } x'$$

Fig. 9

このような条件を課せばたしかに理論が複雑になり，しかも理論家はプログラミング記法で表されるすべてのプログラムが健康条件を満足することを証明しなければならなくなります。皆さんは，述語「偽」はそのような健康条件を満足せず，したがって，その同じ証明によって『理論が値「偽」をその言語で書いたプログラムに結びつけるという間違いを犯さないこと』が保証されることにお気づきかもしれません。

ところで，この作業を行う理由は，プログラムの性質とプログラムを記述する言語について，正しい推論ができるようにすることです。最も単純な推論の方法は，この理論において正しいと証明されている代数等式を使用して記号計算を行うことです。たとえば，任意のプログラムPの前または後にヌル演算IIを合成しても，そのプログラムの観測可能な効果は変化しません。また，合成は結合則をみたします。演算PとQの後に演算Rを行うことは，Pの後にQとRの組を行うことと同じです。最後に，プログラムの合成演算は，非決定的選択“or”を左右両方向から中へ分配できます。(Fig. 10)

LAWS

$$(II, P) = P = (P, II)$$

$$\Big((P, Q), R\Big) = \Big(P, (Q, R)\Big)$$

$$\Big((P \vee Q), R\Big) = (P, R) \vee (Q, R)$$

$$\Big(P, (Q \vee R)\Big) = (P, Q) \vee (P, R)$$

Fig. 10

Prolog

私の講演の中で，最も大胆で驚くべき主張を申し上げるときがきました。それは，Prologプログラムでさえも述語であるという主張です。各述語は，プログラムが実際に実行されたときに起こりうる動作を記述します。今申し上げているのは，Prologを逐次的なプログラミング言語のように見たものであるPrologの手続き的解釈のことです。プログラムは初期状態と最終結果を持っており，その動作はそれら2つの間の関係として定義されます。もちろん，これは論理的な解釈によってプログラムに結びつけられる述語とは全く異なります。Prologプログラムを走らせたときの実際の動作，すなわちその手続き的解釈を記述することに関心があるのです。

PROLOG programs are predicates

Let θ be the initial substitution

Let θ' be the sequence of answers

$\bot$ denotes infinite failure

[ ] denotes finite failure

$$NO(\theta, \theta') \stackrel{\text{def}}{=} (\theta \neq \bot \quad \Rightarrow \quad \theta' = [\,])$$

$$YES(\theta, \theta') \stackrel{\text{def}}{=} (\theta \neq \bot \quad \Rightarrow \quad \theta' = [\theta])$$

Fig. 11

Prologプログラムの初期状態は代入であり，それは，関心の対象となる各変数に，その変数が取ることになっている最も一般的な形の値を表す記号式を割り当てるものです。そのような代入は，一般にθと呼ばれます。Prologプログラムの結果θ'は，従来の言語の結果とは異なります。それは単一の代入ではなく，解代入の列で，要求に応じて次から次へともたらされるものです。

ここで，すべてのPrologフログラムの中で最も単純な2つのプログラムを定義することができます。1つは，有限時間内に必ず失敗するように定義されるプログラムNOです。このプログラムを正常な初期状態θで起動すると，もたらされる結果θ'は答の空の列となります。(Fig. 11)

もう1つは，プログラムYESです。このプログラムを非失敗状態θで起動すると，起動時の初期状態θと全く同じであるけれども，唯一の要素を持つ列として包まれた答を返します。(Fig. 12)

append(X, Y, Z)

Let $\theta =$ “$Z = [1, 2]$”

Then $\theta' =$

“$X = [\,]$,$Y = [1, 2]$,$Z = [1, 2]$”

“$X = [1]$,$Y = [2]$,$Z = [1, 2]$”

“$X = [1, 2]$,$Y = [\,]$,$Z = [1, 2]$”

Fig. 12

もっと実質的な例を挙げてみましょう。お馴染みのPrologプログラム，APPENDです。このプログラムを，Z＝〔1，2〕という初期状態θで起動したとしましょう。APPEND(X，Y，Z)を

この初期状態に適用すると，3行にわたる表に示されている答の列が得られます。最初の答では，Xは空の列を取り，Yは列全体を取り，Zはもとのままです。2番目の答では，この列の2つの要素がXとYの間で分割されます。3番目の答では，Xが列全体を取り，Yは空となります。(Fig. 13)

PROLOG or (;) is append

$$(P;Q)(\theta,\theta') \stackrel{\text{def}}{=}$$

$$\exists X,Y.\ P(\theta,X)\ \ \&\ \ Q(\theta,Y)$$

$$\&\ \text{append}\ (X,Y,\theta')$$

where append $([\bot],Y,Z)$

append $([\,],Y,Y)$

append $([X|Xs],Y,[X|Zs])$

if append (Xs,Y,Zs)

Fig. 13

Prologの OR(逐次的OR)の効果は，最初のオペランドが作り出す答の列を2番目のオペランドが作り出す答の列の前に付加するだけで得られます。PとQは，それぞれ同じ初期状態θで起動されます。プログラムPは答Xを生成し，プログラムQは答Yを生成します。(P ; Q)によって生成される答は，XとYをアペンドしてθ'を作ることによって得られます。APPENDの定義は，初期列が無限失敗である場合に任意の結果を返すことを定義した節を追加したことを除けば，皆様が普段ご覧になる定義と同じです。

優れた数学的な理論では，すべての定義の後に，有用でかつ記憶に残る一群の定理が伴うべきです。たとえば，NOは何も答を出さないので，Pが与える答のリストにそれを追加しても何も変わりません。つまり，NOは，Prologのセミコロンの単位元なのです。同様に，アペンドの結合性はプログラムの合成にもそのままあてはまります。最後に，Prologの逐次的ORは，真に非決定論的なORを通して分配できます。真に非決定的なORとは，私がブール選言として述べたもののことです。(Fig. 14) (Fig. 15)

LAWS

$$NO;\ P\ =\ P\ =P;\ NO$$

$$(P;Q);R = P;(Q;R)$$

$$(P\vee Q);R = (P;R)\vee(Q;R)$$

$$P;(Q\vee R) = (P;Q)\vee(P;R)$$

Fig. 14

PROLOG and (,) is concat

$$(P,Q)(\theta,\theta') \stackrel{\text{def}}{=}$$

$$\exists X,Y.\ P(\theta,X)\ \&\ \text{each}_Q(X,Y)$$

$$\&\ \text{concat}\ (Y,\theta')$$

Fig. 15

Prologの連言は，従来の言語の逐次的合成に非常に類似していますが，単一の結果ではなく結果の列を扱うように体系的に修正したものとなっています。最初の引数Pが生成する列Xの中の各結果は，2番目の引数Qの起動のための初期状態となります。その結果生成される列は，すべてconcat関数で1つに連結されます。形式的定義はいくぶん複雑ですが，非常に単純な代数法則に従います。(Fig. 16)

LAWS

$$(YES,P)\ =\ P\ =\ (P,YES)$$
$$(P,Q),R =\ P,(Q,R)$$
$$(NO,P)\ =\ NO$$
$$NO\ \Rightarrow(P,NO)$$
$$(P;Q),R\ =\ (P,R);(Q,R)$$
$$(P\vee Q),R\ =\ (P,R)\vee(Q,R)$$

Fig. 16

Cut* is truncation

$$P!(\theta,\theta') \stackrel{\text{def}}{=}$$

$$\exists X.\ P(\theta,X)\ \ \&\ \ \text{trunc}(X,\theta')$$

where trunc $\Big([\bot],Y\Big)$

trunc $\Big([\,],[\,]\Big)$

trunc $\Big([X|Xs],[X]\Big)$

* slightly simplified

Fig. 17

最初の法則は，答YESが逐次的合成の単位元であり，プログラムの動作には全く関係ないことを示しています。予想されるとおり，逐次的合成は結合則をみたし，零元NOを持ちます。しかし，これは左零元でしかないことに注意してください。Pが無限失敗をおこす可能性があるので，逆の法則は成り立ちません。

最後に，逐次的合成は，Prologの逐次的ORだけでなく選言つまり非決定性を通して左向きに分配できます。しかし，右向きの分配という相補的な法則は成り立ちません。(Fig. 17)

私たちの定めたPrologの手続き的意味の試金石は，ここでは若干単純化した形で扱いますが，CUTなどの，言語のいわゆる非論理的機能を扱う能力です。カットされたプログラムは，高々1つの結果，すなわちプログラムがいずれにせよ生成する最初の結果を生成します。この結果は，列を打ち切ることで得られます。打切り演算子は，無限失敗と有限失敗の両方を保存するように定義され，失敗以外の状況では，引数が生み出した答のリストの中から余分な要素をただ削除します。CUTの動作を記述する法則も本当にずいぶんと単純です。最初の法則は，カットはベキ等であることを示していて，もともと高々1つの答しか生成しないプログラムをカットしても，観測可能な動作は変わらないという明らかな事実を述べています。(Fig. 18)

LAWS

$$P!! = P!$$

$$(P;Q)! = (P!;Q!)!$$

$$(P,Q)! = (P,Q!)!$$

$$YES! = YES \qquad NO! = NO$$

Fig. 18

次の2つの法則は興味深いものです。2つのプログラムを合成したものから結果を1つだけ得たいとすると，多くの場合，その構成要素から結果を1つだけ計算するだけで十分であることを示しているからです。たとえば，PrologのOR(セミコロン)をカットしたらまずPとQという2つのオペランドをカットしたのと同じこと。同じことは，逐次的連言でも2番目のオペランドについては当てはまります。もちろん，以上の法則は，カットが絡むプログラムの実行効率を大幅に高めるために使用できます。最後に，YESとNOは，両方とも答を1つしか生成しないので，カットしても変化がないことは明らかです。(Fig. 19)

Negation is **not** illogical

$$\sim P(\theta,\theta') \stackrel{\text{def}}{=}$$

$$\exists Y.\ P(\theta,Y)\ \&\ \text{neg}\ (\theta,Y,\theta')$$

where

$$\text{neg}\ \Big(\theta,[\bot],Z\Big)$$

$$\text{neg}\ \Big(\theta,[\,],[\theta]\Big)$$

$$\text{neg}\ \Big(\theta,[Y|Ys],[\,]\Big)$$

Fig. 19

Prologの否定は，カットに比べて扱いが難しいことはありません。この場合も，単純なリスト演算NEGをそのオペランドが生成する結果に適用するだけです。NEG述語は，無限失敗を保存し，有限失敗を成功に変え，また他のすべての場合には有限失敗をもたらすと定義されます。(Fig. 20)

LAWS

$$\sim YES = NO \qquad \sim NO = YES$$

$$\sim P\ =\ (\sim P)!\ =\sim (P!)$$

$$\sim\sim\sim P = \sim P$$

$$\sim (P;Q)\ =\ (\sim P, \sim Q)$$

$$\sim \text{true}\ =\ \text{true}$$

$$\mu X.\ \sim X\ = \text{true}$$

Fig. 20

真理値のProlog否定を支配する法則は，YESとNOの場合についてはブール否定の法則と同じです。二重否定に関する古典論理の法則は成り立ちません。それは，直観主義論理の三重否定の法則に弱めなければなりません。

最後に，ブール代数におけるおなじみのDe Morganの法則の中の1つと驚くべき相似関係にある法則があります。否定は，逐次的ORを通りぬけて分配でき，それを逐次的ANDに変更します。明らかに，ANDの式はORの式よりも計算がずっと速いので，これは最適化に有効かもしれません。しかし，これと双対のDe Morganの法則は成り立ちません。

これで，Prologの基本構造に関する私の簡単な説明を締めくくらせていただきます。これらのすべての構造は，無限失敗がない限り，どんな初期代入θに対しても，プログラムが生成しうる解代入の列θ'は高々1つであるという意味で決定的です。しかし，プログラムを述語として解釈する大きな長所は，単純な方法で非決定性を導入できることにあります。(Fig. 21)

Or − Parallel

$$(P\|Q)(\theta,\theta') \stackrel{\text{def}}{=}$$

$$\exists X,Y.\ P(\theta,X)\ \&\ Q(\theta,Y)$$

$$\&\ \text{inter}\ (X,Y,\theta')$$

$$\text{where inter}\ ([\bot],Y,Z)$$

$$\text{inter}\ ([\,],Y,Y)$$

$$\text{inter}\ ([X|Xs],Y,[X|Z])$$

$$\text{if inter}\ (Xs,Y,Z)$$

$$\ldots$$

Fig. 21

LAWS

$$(P\|Q)\ =\ (Q\|P)$$
$$P\|(Q\|R)\ =\ (P\|Q)\|R$$
$$P;Q\ \Rightarrow\ (P\|Q)$$

because

$$\text{append}\ (X,Y,Z) \Rightarrow \text{inter}\ (X,Y,Z)$$

Fig. 22

たとえば，大勢の研究者が，Prologの逐次的ORの改善を提案してました。逐次的ORに対する改善の1つは，ブール選言のように可換にすることです。もう1つの改善は，両方のオペランドを並列に実行し，結果を任意にインタリーブすることを許すというものです。これら2つの長所は，並列ORの定義によって同時に実現できます。並列ORではPとQが生成するXとYという結果は，互いにアペンドするのでなくインタリーブされます。(Fig. 22)

さて，並列ORは，逐次ORの代数法則の多くを保存するとともに，対称でもあります。アペンドは，単にインタリーブの特殊な場合にすぎませんから，APPEND(X, Y, Z)がINTER(X, Y, Z)を含意することがわかります。その結果，逐次的ORは，並列ORの特殊なケースであるとともに，常にその正当なインプリメンテーションとなります。逐次的ORは，並列ORよりも決定的です。逐次的ORの方が予想と制御が簡単で，しかも並列ORが満足するすべての仕様を満足します。要するに，逐次的ORは，あらゆる点で，あらゆる状況において，そしてあらゆる目的に関して，並列ORよりも優れています。あらゆる点と申しましたが，並列マシン上での実現が低速かもしれないという，ただ一点を除きます。私の言う，ブール選言で表現される非決定性は，本質においてデーモン的です。プログラマは，マシン(つまりデーモン)が常にプログラマの好みとは反対の選択をすると想定しなければならないので，プログラミングは決して容易にはなりません。しかし，この例を含めて多くの場合，非決定性は仕様と設計を単純化し，それらをより高い抽象レベルで推論するのを助けます。

KL 1

この講演の冒頭で，Prolog言語を最初の設計者と論理プログラミングの発展から私が学ばせていただいたことは賞賛に値すると申し上げました。この講演を締めくくるにあたり，第五世代コンピュータシステムプロジェクトで設計，実現および

使用されてきたKL 1 言語の設計者たちから学ばせていただいたことを要約したいと思います。

イギリスのインモスで，David Mayを始めとする研究者たちが10年前に設計したOccamプログラミング言語とKL 1 言語を，簡単に比較してみようと思います。1982年 9 月，私は彼らとともに日本を訪問し，Occamを当時結成されたばかりのICOTに披露しました。KL 1 言語と同じように，Occamもトランスピューターとして知られる新世代の並列コンピュータ上での並列実行に合わせて設計されたものです。トランスピューターは，基本的には，小規模な計算機ネットワーク上で動く組込み型実時間アプリケーションに対応するように設計されました。このためには，プロセッサと記憶領域を論理的プロセスに静的に割り当てて，インプリメンテーションの最高度の効率性を実現することが必要でした。皆さんのKL 1 言語はその10年後に設計されたのですが，その頃には何百台ものマシンからなるネットワークが実現可能になりました。これは，最初からプロセッサの動的割当てのオーバーヘッドを受け入れなければならないということです。皆さんのKL 1 言語は，範囲は異なるものの，同じように挑戦的なアプリケーションのために設計されました。それは高度に不規則で，動的に生成されてゆく記号データ構造の処理が必要になるようなアプリケーションです。つまり，最初から記憶領域の動的な割当てと再割当てのオーバーヘッドを受け入れなければならないことを意味しています。これらのことが，KL 1 とOccamが根本的に異なる主な理由だと思います。しかし，この 2 つの言語では，質的な類似点の方がいっそう顕著です。どちらの言語も小さく単純なので，専門的なプログラマでなくとも，また他の領域の専門家でもすぐに学んで使用することができます。しかし，記憶領域の動的割当て機能があるのでKL 1 の方がより単純です。

どちらの言語も効率が高く，システムの個々のプロセッサの能力を最大限に活用することができます。しかし，記憶領域を静的に割り当てるので，Occamの方がやはりいくぶん効率が高いでしょう。両言語とも，操作的な解釈と独立の明確な抽象的意味を持っています。このため，構造やアーキテクチャが大幅に異なる，複数の並列マシンでのインプリメンテーションの互換性を完全に保証することができます。

両言語の意味は，実行メカニズムの実際の動作を記述する述語を用いて表現できます。これによって，現実の世界のユーザの要求を記述したより抽象的な述語として表現された仕様から始まる，開発への確実な道が開かれます。意味論から，エレガントでわかりやすい代数法則をいくつも導くことができます。そのような法則は，人間の知性と理解の助けになるだけではありません。特定の実行メカニズムのアーキテクチャ上の特徴に合わせて行なうプログラムのローカルな最適化やグローバルな再構造化の両方について，その正当性を保証してくれます。私だったら，FORTRANプログラムから存在しないと思われる並列性を見つけようとするよりは，非決定的なKL 1 プログラムを最適化する方を選びます。

どちらの言語も，高い並列度を引き出すことを助長し，また粒度を細かくとることもできますが，KL 1 の方がOccamよりも，粒度を一段と細かくとることができます。粒度を細かくとることによって，システム内の個々の物理プロセッサが，多数の論理プロセスを確実に時分割処理できるようになり，ユーザは，時分割処理が行われているかどうか心配する必要がなくなります。これは並列スラックネスと呼ばれ，マシンの高い利用度を保証するとともに，メッセージの転送待ち時間やキャッシュ・ミスが実行効率に与える影響を隠してくれます。

プログラムの実行を逐次化して非決定性の度合いを注意深く制御することによって，さらに効率を高めることができ，転送待ちによる遅れをさらにへらすことができます。OccamとKL 1 の両言語とも，並列プロセスによる共有記憶の更新という，言語に絶する恐怖から，プログラマを完全に保護してくれるという，非常に重要な特徴を備えています。

最後に，両言語とも，新しい世代の現実の並列プログラミングアーキテクチャの設計と実現を促してきました。それは，トランスピュータの諸リリースの一族であり，各種の形態とアーキテクチャの並列推論マシンの一族であります。これらのマシンが十分な生の計算パワーを提供してくれたおかげで，新たな潜在的ユーザたちが新しいクラスの問題を解くために，新しいプログラミング手法を学ぶ気になってきたのです。

以上，私がKL 1 の中から発見した技術的な長所について手短かに概観してみました。KL 1 と皆様の輝かしい前途を心からお祈りするとともに，その到来を予言したいと思います。しかし，将来は，多くの偶然的，商業的，政治的，および経済的な，私の知識と能力の及ばない各要因に左右されます。技術的な業績の大きさは立派なものであります。業績の大きさは，批評家や悪口を言う人の懐疑，不信，はては嘲笑によってさえも，少なくとも部分的には測ることができます。「そんなことはできないだろう」と言われれば言われるほど，実際にそれを成し遂げてきた皆様の業績は大きくなると思います。

皆様は，技術的および科学的に高い質の洞察，決断および発明に加えて，大きな，正真正銘の勇気を示してこられました。皆様の勇気に感謝を捧げるとともに，皆様の業績を世に知らせるという趣旨で開催された，この最終会議の最終日のこの最後のセッションにおいて，皆様の前で講演をするようご招待いただいたことに改めて感謝いたします。ご静聴ありがとうございました。

－C. A. R. Hoare教授略歴－

Hoare教授は，1956年にオックスフォード大学で古典語であるラテン語とギリシャ語の学位を取得されました。同教授は，今日に至るまでプログラム言語に関心を寄せていらっしゃいます。卒業後は，まずコンピュータメーカーに勤務され，そこで最初に取り組まれた仕事がALGOL 60のインプリメントでした。コンピュータ業界で 8 年間過ごされた後，ベルファストにあるクイーンズ大学の教授に就任されました。そこでは，プログラミングの論理的な基礎とプログラミング記法の研究を行われました。1977年にオックスフォード大学の教授に就任されてからは，それまでの基礎研究の成果のいくつかを，実際的で，かつ採算の取れる産業界のアプリケーションに移転することを始められました。教授は，これまでに手続き型プログラミング，オブジェクト指向プログラミング，および並列プログラミングの分野で大きな業績を残されています。OccamとTransputerは今や非常に有名です。

質問：ANDとNOTの話の最初のところで引合いに出された例に感心しました。仕様のたぐいは，温度と圧力を制御しなければならないとか，爆発してはならないとかいうことが表現できないといけないとおっしゃいましたね。でも，この種の文献でこれまでに目にしたものから考えて気がかりなのは，因果関係の存在する現実の世界とどのようにインターフェースをとるのかということには，ほとんどふれていないのではないかということです。現実の世界には，状況を変化させている他のエージェントが存在します。そして，誰かが状況に干渉するようなことをしたら，明らかに爆発が起きるかもしれません。そこでこの考え方が，現実世界を本当に制御したりそれと相互作用を行ったりしなければならないプログラムと，どのように調和するのかについてコメントをいただけると

ありがたいのですが。

Hoare：ご質問ありがとうございます。数学的および科学的理論と，現実の世界で観測できるまたは観測したいものとの関係を確立するという問題は常につきまといます。それは，今後何年にもわたって哲学者を悩ませ続けるであろう問題です。そのようなきわめて重要な関係を確立するのに，私の知る限りでは，非形式的にやるしか方法がありません。形式的メカニズムでは，決してその関係の確立はできないということです。非形式的メカニズムとは，獲得した要求を最初に形式化したものが，制御対象となるプロセスの物理的現実に確かに対応することを，領域の専門家の理解にたよって確認するというものです。リアルタイムプロセス制御システムの形式的仕様を，私は，どちらか言えばプラントの記述，つまり物理的条件や，プラントの動作や，プラントの故障のしかたの記述として見ています。これらのことがわかっていれば，プログラムの動作は，適切な，あるいは，最適な，または少なくとも許容できると考えられるプラントのすべての状態の逆像ないしは鏡像によって指定できます。

私はプロセス物理学者ではありませんので，このレベルで形式的にとらえた仕様の妥当性をチェックできるとは思いません。しかし，私は，クリティカルなプログラムの開発プロセスの形式化は，このレベルで始まるべきであると思っています。制御対象の物理的現実を理解している人が定めた正確な仕様を，計算機科学者およびソフトウェアエンジニアが，そのままもちつづけて，クリティカルなプログラムの開発にまで結びつけることができるような道を探らなければなりません。この会議の最初の招待講演で，Dines Bjorner教授が，ヨーロッパでPROCOSとして知られるプロジェクトにおいて，このような手法を開発することへの私たちの共通の関心について述べていらっしゃいます。

質問：OccamとKL 1という2つの並列プログラミング言語を比較されましたが，現在研究されている並行処理に対する別のアプローチについてコメントしていただけますか。

Hoare：並行処理に対する他のアプローチについてコメントして欲しいということですが，これについては一晩使ってお話ししたいくらいです。

並行処理(concurrency)の問題と高並列コンピュータの問題は，依然としてつきまとっています。世界に対して，新しいプログラミングパラダイムと新しいハードウェア構造を同時に採用するよう説得するというのがどういうことか予測するのはきわめて困難です。世の中は，苦い経験から，前のコンピュータが実行したのと全く同じプログラムを新しいコンピュータでも実行できることが最も重要であることを学んできました。そのため，コンピュータメーカーは，新しいプログラムの実行が必要となるコンピュータアーキテクチャの市場を開拓することができないのです。私には，その結果がどうなるか予測するような才能はありません。ただし，プログラマは，非決定性のためにプログラムのテストが不可能であるとわかると，新しい技術やマシンばかりでなく，新しい能力までも即座に拒否するだろうと確信しています。非決定性に対するこれらのマシンを安心して使用する確実で科学的な制御を提供するOccamやKL 1などの言語を使用しなければ，プログラムのテスト中には現れないで，後でユーザを悩ませるようなエラーがマシンに入っていないことを確認するのは，なんと言っても専門的なプログラマの仕事です。そのようなエラーに対する対処法や保護を提供しない手法や言語でプログラムを作成するときめた人々は，結局はそのようなエラーがつみ重なって破滅に至ることを悟るだろうと確信しています。

質問(岡田)：簡単な質問だといいのですが。教授は論理的な概念である述語と，プログラムとをまとめられました。ことによると述語のような伝統的学説を拡張して。たとえば，仕様は述語であります。教授は，プログラムは述語であるとおっしゃいました。では，教授が示された論理含意の正当性の証明を考えましょう。プログラムとしての述語が仕様を含意するというような形のものです。すると，それは論理的ないしは数学的な概念であり，またプログラミング言語理論ないし計算機科学の何らかの概念に関連しています。ではその場合，証明に対応するものは何なのでしょうか。つまり，教授は正当性を証明していらっしゃいます。論理的含意の証明です。では，その証明は何によって解釈されるのでしょうか。教授の枠組では，証明とは計算機科学的な意味において何なのでしょうか。

Hoare：ありがとうございます。残念ながら，証明論に関する知識は全く持っておりません。私が考えている証明というのは，従来の数学的証明で行われている慣習にできるだけ近づけたもので，論理学者や集合論学者によって明らかによく形式化されているものです。数学的証明と計算は人間によってかなり確実に行うことができるし，また他の人間によってさらに確実にチェックすることができると思います。したがって私は証明に対して，形式的または機械的なチェックの可能なアクティビティとして，多大の注意を向けてきたことはありません。設計段階での証明は，数学の教科書で見られるような種類の証明と全く変わらないものだとみなしています。

岡田：たとえばHoare論理の立場から言いますと，教授は以前の研究で，プログラムの正当性の証明体系を指定されました。それがいわゆるHoare論理です。教授の現在の理論に対しては，そのような類の形式的な枠組はもはや考えられていないのでしょうか。

Hoare：全くその通りです。現在では，プログラムの正当性をきちんと記述し，確立するのに，Hoare論理よりもずっと優れた方法があると思います。Dijkstra教授が最弱前提条件(weakest pre-conditions)に関する研究を発表された1974年以降は，Hoare論理は時代遅れになってしまったと考えています。