パネル討論:

理論は実践を導けるか、実践は理論を生かせるか？

司会:　萩谷昌己(京大)

パネリスト:
(理論代表)　大堀淳(沖電気)、柴山悦哉(龍谷大)、二木厚吉(電総研)
(実践代表)　上田和紀(ICOT)、竹内郁雄(NTT)、村井純(慶応大)、安村通晃(慶応大)

1980年代を通じ、関数型・論理型プログラミングと並行・並列処理の分野を中心として、プログラミングのための基礎理論は大きな進展を遂げています。また、コンパイラ生成系、オブジェクト指向言語、分散OSなど、応用技術も大きく進歩しています。しかし、基礎理論の研究者はより深い理論を追い求め応用を忘れるという傾向にあり、一方、応用技術の方でも、基礎理論は役に立たないものだという決めつけが感じられます。本パネルは、このような基礎理論と応用技術の乖離をどうのように考えるべきか、基礎理論と応用技術の溝を埋めることは可能か、ということについて議論していただき、1990年代におけるプログラミングの理論と応用の研究を方向付けていただきたいと思って開催するものです。

Panel Discussion:

Can theory lead practice? Can practice use theory?

Charied by:　Masami Hagiya (Kyoto University)

Panelists:
(Theory)　Kokichi FUTATSUGI (ETL), Jun OHORI (Oki Electric Industries),
Etsuya SHIBAYAMA (Ryukoku University)
(Practice)　Jun MURAI (Keio University), Ikuo TAKEUCHI (NTT),
Kazunori UEDA (ICOT), Michiaki YASUMURA (Keio University)

Throughout 1980's, there have been made great progresses in foundational studies on programming, particularly in the fields of functional/logic programming and concurrent/parallel processing. On the other hand, significant contributions have also been made in the fields of practical applications, such as compiler generators, object-oriented languages, distributed operating systems, etc. However, researchers in foundational studies tend to persue deeper and deeper theory and forget practical applications, while in the communities for practical applications, it is widely considered that foundational studies are of no use. In this panel discussion, panelists are expected to talk about this gap between theory and practice, discuss a means for filling the gap, and finally give research directions in 1990's in both theory and practice on programming.

パネル討論：理論は実践を導けるか、実践は理論を生かせるか?

プログラム言語における理論の役割

大堀 淳　　沖電気工業（株）関西総合研究所

1　はじめに

プログラムの理論的研究の目的の一つは、信頼性のあるソフトウェアシステムを効率的に開発するための組織的方法を確立することである。この意味で「理論」は、「実践」のもっとも基本的なステップの一つである。このような理論の役割は、複雑さを増す情報システムに対処する上で、いっそう重要なものとなると予想される。

筆者は、「理論」と「実践」の間に深刻な乖離があるとは考えていない。特に最近のプログラム言語の理論的研究の中には実践を念頭においたものも数多くあり、実用上有用な技術が多数開発されている。しかしながら、理論研究が注意を払っていない重要な分野も多く存在するのも事実である。そこでパネルでは、筆者の関心を持つプログラム言語の型システムの分野で、理論と実践の関係の問題点を、理論研究の観点から議論する。

2　型システムにおける理論の役割

プログラム言語の型システムは、プログラマの書いたコードがどのようなプログラムとして処理系に認識されるかを規定するシステムである。現代のプログラム言語は、複雑な情報処理システムを効率良く構築するため、高階の関数、汎用の手続き、データ抽象等々の高度な機能を多数含んでいる。型システムは、それら複雑な諸機能を一つのシステムに統合しなければならない。

それら種々の機能は複雑な相互作用を持っており、整合性のある型システムを設計するためには、形式的な手法によるプログラムの数学的性質の分析、すなわち「理論」が不可欠である。この観点から特筆すべき実用プログラム言語に、Standard ML がある。その型システムは、Milner の型推論理論 [5] に基づき設計されている。この理論では、プログラムの実行が型判定を保存することが厳密に証明されている。この性質によって、「型チェックされたプログラムは、実行時に型エラーを起こさない」という性質が任意の複雑さを持つプログラムに対して保証される。この高い信頼性は、有識者からなる委員会による決定などからは、得ることが難しいと思われる。

1987年、Robin Milner 率いるＭＬチームが英国コンピュータ学会の賞（the British Computer Society's 1987 Award）を授賞した [4]。AT&T の MacQueen らが実装した実用コンパイラー Stadnard ML of New Jersey とともに、これは、理論研究が実践を導き得ることを示す鄭かしい成果といえよう。

3　理論を実践に生かす上での問題点

しかし反面、実用上価値の高い理論研究が賞の対象となること自体、理論的研究が必ずし実践の基礎となり得ていない事の反映と見ることもできる。

「理論」側の問題点の一つとして、その研究の対象が必ずしも実践上の問題を反映していなかったことがあげられよう。例えば二階ラムダ計算の理論は、Domain Theory やカテゴリー論を用いた種々の美しい意味モデルを生み出したが、その関心の対象は主に関数型であり、レコードやバリアントなどを含んだシステムの性質の研究は著しく遅れていた。これは、それらデータ構造が実用言語のもっとも基本的なものであることを考えると、いささか驚くべきことである。

4　実践に生かし得る理論のために

幸いに以上の傾向は変わりつつあるように思える。変化を示す例としては、ここ数年来の、レコードのための型理論の研究があげられる [8,3,6,7,1,2]。これは、理論的研究が実用的なシステム構築の基礎としても、潜在的に高い可能性を持っていることを示唆している。

理論研究が、オブジェクト指向プログラミング、データベース、分散処理、マルチメディア通信などの分野でも、有用性の高いアルゴリズムや新しい言語を生み出すことを期待したい。理論研究がその対象を拡張することはまた、新しい手法や概念が生み出され、理論研究にとっても好ましいインパクトを与えるはずである。

参考文献

[1] L. Cardelli and J. Mitchell. Operations on Records. *Mathematical Foundation of Programming Semantics*, 1989. [2] R. Harper and B. Pierce. A record calculus based on symmetric concatenation. *POPL*, 1991. [3] L. A. Jategaonkar and J.C. Mitchell. ML with extended pattern matching and subtypes. *LISP and Functional Programming*, 1988. [4]LFCS Anual Report. University of Edinburgh. 1987-1988. [5]R. Milner. A theory of type polymorphism in programming. *JCSS*, 17, 1978. [6]A. Ohori and P. Buneman. Type inference in a database programming language. *Lisp and Functional Programming*, 1988. [7]D. Remy. Typechecking records and variants in a natural extension of ML. *POPL*, 1989. [8]M. Wand. Complete type inference for simple objects. *LICS*, 1987.

パネル討論: 理論は実践を導けるか、実践は理論を生かせるか?

プログラムの理論とプログラミングの理論 — 理論、実践、そして言語 —

柴山悦哉 (龍谷大学理工学部)

現在までに、プログラムの検証などに関していくつかの研究をおこなってきた。そこで、本稿ではプログラムの理論とプログラミングの理論に関して思っていることを述べることにする。

理論と客観性

そもそも理論は何を目的とするのであろうか。様々な現象を客観的な形式で記述し人々の理解を助けるのが理論の役目であると、本稿では考えることにする。感の良い人(天才)なら非言語的に知っている事柄を、感の悪い人にもわかるように言語化する作業が理論の研究だと言ってもよいだろう。この定義がうまく機能していれば苦労はない。理論は実践を導けるはずだし、実践は理論を生かせるはずだ。ところが現実は厳しい。

理論に対する第1の要請は客観性であり第2の要請は了解性である。客観性を捨てると理論とは認められない。客観性を追求するためには、了解性を捨てることさえも許される。それほどまでに、客観性は重視されている。

数学という記述系

多くの学問分野において、理論は、数学という記述系を用いて表現される。物理学、経済学などはその代表例であろう。自然言語に比べ数学の方が客観的記述に向いているから、これは当然の傾向と言える。プログラムの理論、プログラミングの理論も数学によって記述されることが多い。

ここで問題となるのは、ソフトウェア界の現象を記述するための言語として、数学は本当に有効かどうかという点である。数学が、物理現象を記述するにあたって大きな成功をおさめたことは否定できない。しかし、プログラムの理論が対象とするのは物理的実体を持たない論理的な存在である。プログラミングの理論は人間の認知的な過程を対象とする。少なくとも、プログラムやプログラミングを対象とした時、大学の初年級で習う解析学や線形代数学の手法はほとんど役に立たない。

客観的な記述系としては数学が一番確実なので、数学にたよって理論を構築しようとする。これで、本当に良いのであろうか。たとえば、人間の認知過程としてのプログラミングという現象を記述する能力が数学にあるかどうか、はなはだ疑問を感じる。

プログラムという対象

プログラムは離散的な存在である。しかも、連続体力学で成功したような方式で近似的に連続モデルを作ることもできそうにない。なにしろ、あるプログラムのソースコードや入力データを1文字書き換えただけで、プログラムの挙動は大きく変わる可能性がある。だから、近似モデルを作るのは困難だと思われる。

近似モデルを作れないということは、これは大変なことである。物理学の計算でよく行なわれるような微小な項の切捨てもできないし、実験結果に対して最小2乗法を適用するような意味での近似もできない。可能なのは、与えられたプログラムの当面興味がない性質を見ないようにすることだけである。

さらに都合の悪いことに、問題をはらんだプログラムはたいてい複雑だという経験的事実がある。10行程度の短いプログラムなら、理論とか実践とか細かいことを言わなくても、どうせ何とかなるのである。複雑なものに対して、exact なモデルを作るのは、至難の技と言えよう。

もう一つの記述系

「客観的記述」という大原則は曲げられない。偏微分方程式などの実践的数学も使えない。では、どうすればよいのか。ソフトウェア界で独自の記述系を考案しなければなるまい。こう書くと大変なことのように聞こえるが、そうでもない。我々の手には、プログラムという客観的記述系がある。しかも、石頭の機械にでも解釈できるくらいに了解性は高い。

数学の基礎付けをメタ数学としての数学基礎論が行なうように、プログラムの基礎付けにメタプログラムを使うのは正しいアプローチではあるまいか。実際、現実的なプログラミング言語の厳密な意味は、メタプログラムとしてのコンパイラやインタプリタによって与えられている。この現状に対して批判もあるが、他に良い方法があるわけではない。アルゴリズムの理論にしても、オーダーの解析には数学を使うが、アルゴリズムの記述自体は(疑似)プログラムによって与えられる。検証などの面倒な問題も数式で表現できたものではなく、プログラムで表現されたものだけを評価すればよいだろう。

理論と実践

こう考えると、理論と実践の違いがわからなくなる。たとえば、500頁にわたる数学の証明と同じ頁数のソースコードを目の前にした時、前者が理論で後者が実践の成果だと考えるべきなのだろうか。あるアルゴリズムに関する論文を書くのと、そのアルゴリズムを実現したライブラリを広く世間に公開するのは、世の中への貢献としてほとんど同格なのではなかろうか。

結局重要なのはユーザから見た切口だろう。たとえ500頁にわたる証明を持っていても、定理の記述自体が簡潔で了解性が高くしかも応用の効くものなら、それで良いのだ。プログラムのソースコードにしても同じだろう。プログラム自体は複雑でも、ユーザインタフェースが良くできていて、使い易く有用であれば、それでいい。

そして言語

最後にプログラミングの理論に関して一言コメントしよう。人間の認知過程をまともに理論化しようとしても、現在の技術レベルでは荷が重かろう。最大の現実的貢献は、良いプログラミング言語を設計することではなかろうか。言語設計を理論と呼ぶのは問題も多いが、計算モデルが持つべき重要な性質を指摘するのは、理論の課題だと思う。

新研究会の名前にちなんで言うならば、「基礎」も「実践」も所詮は「言語」によって記述する作業である。「基礎」と「実践」の要請を満たす良い「言語」を作っていきたいものである。

[パネル討論: 理論は実践を導けるか，実践は理論を生かせるか?] 形式仕様の理論と実戦

形式仕様の理論と実践

二木 厚吉
電子技術総合研究所
futatsugi@etl.go.jp

はじめに

形式仕様(Formal Specifications)は，ソフトウェア開発に関連した研究テーマの中で最も理論的と見なされるものであろう．さらに形式仕様は(ソフトウェア開発のための)形式的な方法(Formal Methods)の基盤であり，形式仕様の理論と実践は取りも直さずFormal Methodsの理論と実践である，と言う側面も持つ．本稿では，「形式仕様の理論と実践」に関する基本的な問題点を概説することで，計算機科学における理論と実践の望ましい協調を実現するために何が必要なのかを考える材料を提供したい．

以下の議論を混乱させないためにも，筆者の立場は「形式仕様並びにそれに基づくソフトウェア開発の形式的な方法は，理論に導かれかつ実践に生かすことが可能な技術である」というものであることを，あらかじめお断りしておく．

形式仕様と非形式仕様[1]

「仕様」の定義として辞典には「特定の形状，構造，寸法，成分，精度，性能，製造法，試験方法などの規定」とある．仕様の重要な性質としてもう一つ付け加えるならば，それが不特定多数の人に読まれることを目的とした文書である点である．仕様はそれに基づいて具体的なものやシステムを作る際に最後の拠り所となる文書であり，次のような性質を持つことが要求される．

- 無矛盾性: 規定されている内容に自己矛盾がない．
- 完全性: 必要な内容がすべて規定されている．
- 非曖昧性: 規定されている内容が一意に特定できる．
- 明晰性: 規定されるべき内容が理解し易く記述されている．

仕様を記述すべき内容が「机の高さ」といった万人の常識に属すものであれば，単に65.0cmといった数値を記すだけで仕様を与えることができる．しかし，情報通信システムや計算機の操作システムといった複雑なシステムの仕様の場合には，仕様の書き方自体が大きな問題になる．

仕様には次の2つの種類があることが徐々に認識されるようになってきた．

- 非形式仕様(Informal Specification):自然語で書かれた仕様のように，意味を定める厳密な規則が利用可能な形で与えられておらず，その仕様を取り巻く常識や技術体系といった状況に依存してしか意味を定めることができない仕様．
- 形式仕様(Formal Specification):あらかじめ与えられた利用可能な厳密な規則に従い意味を定めることができる仕様．

複雑なシステムの仕様は自然語に図や表を交えて非形式的に記述するのが普通である．しかし，信頼性の高いソフトウェアシステムを効率的に開発するための要求仕様や設計仕様の記述法の開発を目的とした「ソフトウェアの仕様記述法」の研究を通じて自然語で書かれた非形式仕様の次のような問題点が認識されるようになった[7].

すなわち，自然語で書かれた非形式仕様では，その仕様の作成者の常識やその仕様の背景となっている技術に関する知識などに依存した記述が暗暗裡なされる可能性が大きく，無矛盾性等の性質を判定する際の基準となるべき「仕様の意味」を客観的に特定することが難しい．従って，複雑な

システムの自然語で書かれた仕様が無矛盾性，完全性，非曖昧性などの性質を満たすかどうかを客観的に判定することが極めて困難である．

複雑なソフトウェアシステムの開発における仕様の果たす役割の重要性に対する認識が高まるにつれて，開発すべきシステムの色々な性質を仕様のレベルで解析できる可能性の高い形式仕様に基づくソフトウェア開発法への関心が高まってきている．

形式仕様の仕組み: 理論が提供すべきもの[1,9]

形式仕様の最大の特徴はその意味をあらかじめ与えられる規則にしたがって厳密に定めることができるという点である．形式仕様の意味を定義する規則は，通常次のような要素から構成される「形式仕様記述言語」という形で与えられる．

(1) 構文規則: どのような記号の並びが仕様文書であるかを定める構文規則．通常形式文法(生成文法)で与える．

(2)意味モデルと意味規則: 仕様の意味のモデルを定義する意味モデルの定義と，任意の仕様にその意味に当たる意味モデルを対応させる意味規則．

(3)構造化法: 何を単位にして仕様をモジュール化し，それらのモジュールをどのように組み合わせて仕様を構成するかを定めた規則．

構文規則，意味モデル，構成法が厳密に定められた形式仕様記述言語を用いて仕様を記述しておけば，仕様の意味をこれらを使って客観的に定めることが出来，仕様の無矛盾性，完全性，曖昧性といった性質を判定できる可能性が非形式仕様に比べて飛躍的に高まる．

形式仕様の利点をソフトウェアの開発の中で活かすことを目指して，過去においていくつもの形式仕様記述法／言語が提案されているが，その基本的な性格を定めるのはその意味モデルに何を選ぶかということである．ソフトウェアシステムの仕様記述のための意味モデルを何に選ぶかは，応用領域，仕様記述の目的などに依存する．しかし意味モデルは無矛盾性の判定などに利用できるほどにその性質が分析可能である必要があり，数学または計算機科学の中である程度研究されその基本的性質が解析可能であることが明らかになっている形式的な体系が採用されることになる．

仕様の構造化法は仕様の複雑化/巨大化にともない急速に重要性が高まってきた．形式仕様を明晰に理解し易く記述するためには，効果的な構造化法を採用することが決定的に重要である．構造化法の基本的な性格は，「何を単位にして仕様の構造化を行なうか」により定まり，どのような意味モデルを採用するかに大きく依存する．

形式仕様における意味モデル，構造化法，記述の視点[1]

今までに幾つかの形式仕様記述法/言語が提案されている．ここではそうした形式仕様記述法の基礎にある意味モデルと構造化法の代表的なものを掲げることで，形式仕様記述法の分類のための基準を与える．また，形式仕様記述言語を特徴付ける他の重要な要素としての仕様記述の際の「視点」違いについても述べる．

[意味モデル]

システムの動的な振舞いの仕様記述のためのモデル

- 有限状態機械，遷移システム，拡張有限状態機械
- ペトリネット
- プロセス代数，CCS(Calculus for communicating systems), CSP(communicating sequential processes)

システムの静的な機能や基本構造の仕様記述のためのモデル

- 集合と関数，抽象データ型，多ソート代数系，順序ソート代数系
- 集合と関係，論理系，多ソート一階述語論理系

[構造化法]

・手続き，プロセス，関数を単位とした構造化: サブルーチン，サブプロセス，関数の合成等の概念に基づく
・抽象データ型や抽象オブジェクトを単位とした構造化: クラス，サブクラス，インスタンス，インヘリタンス(継承)などの概念に基づく

[記述の視点]

仕様記述に際しての基本的な視点に関しては次の二つが認められる[9].

・モデル指向(model oriented) 集合，レコード(直積)，項などの数学的な体系(構造)を用いて仕様記述すべき対象を形式化/抽象化した形式的/数学的なモデルを構成する.
・制約/性質指向(property oriented) 仕様記述すべき対象が満たすべき制約や性質を論理式や等式を用いて記述する.

特に通信プロトコルの仕様記述においては次のような記述の視点の相違が認められる.

・システムと外界との相互作用の生起順序の可能なすべての組合せを正規表現で示すなどして，外部から観測可能なシステムの動作を，なるべくシステム内部の機構を規定することなく記述する．例えば，プロセス代数モデルに基づく仕様記述はこれに属する.
・システムを実現する有限状態機械を示すなどして，システムの動作を実現する機構を記述する．例えば，拡張有限状態機械モデルに基づく仕様記述はこれに属する.

形式仕様の研究/実践の状況

形式仕様の研究/実践活動に関してはヨーロッパとアメリカで次のような違いが認められる.

・ヨーロッパ: 理論，方法論中心の研究/実践活動が盛ん．VDM，Zなどの代表的な仕様記述言語はすべてヨーロッパ製．代数仕様に関しても研究/実践が盛ん．OSIの標準化に関連してSDL,Estelle,LOTOSなどの仕様記述言語の標準を定める際に指導的な役割を演じた.
・アメリカ: 高信頼性を要求されるシステム，特に国防関係のシステム，の検証ツールの開発と言う観点からの研究/実践が盛ん．こうした技術は，classifiedであり，禁輸扱いにされている部分もあるらしく，正確な情報は得難い.

形式仕様の本来の目的である．ソフトウェア開発過程の高効率/高信頼化に関しては，ヨーロッパ的なアプローチの延長線上にあるが，実験的な試用を除いては，現場での実践の実績はあまりない．ただし，

・言語システムを中心にしたプログラミング環境における形式化の発展[5],
・OSI関連のプロトコルの標準の開発に関連した活動[1,2],
・VDMやZに関連した活動[3,12],
・代数仕様に関連した活動[4,6]

の状況を見ると現場でも実践出来る部分が明らかになりつつあると言う感じを受ける．形式仕様，形式的な方法の現状の詳細については最近あいつで出版された[12,13]のような雑誌の特集合を参照にするのが良い方法であろう.

形式仕様の利点と欠点[1]

形式仕様は自然語で書かれた非形式仕様の欠点を克服することを目的にして発展してきた．しかし形式仕様にはその基本的な性質から必然的に導かれる欠点も存在する.

形式仕様の利点と欠点を対比すると以下のようになる.

利点:

・無矛盾性，完全性，非曖昧性等の重要な性質が定義しそれらの性質を機械的に判定できる可能性が高い.
・仕様自体の機械処理が可能であり，仕様の解析，蓄積，再利用等に計算機を有効に利用できる可能性が高い.

・形式仕様にはモジュール化/構造化を陽に指示する機能が用意されていることが多いので，仕様文書の構造化/モジュール化がし易い．

欠点:

・その形式仕様記述法が想定している意味モデルから外れた概念の表現は不自然になり易く表現力が制限される．すなわち，非形式現実と形式仕様の不整合，現実世界と抽象世界の不整合などの「現実とモデルの乖離」[9]と言う宿命的な問題が常に存在する．

・特別に設計された形式仕様記述言語の構文規則，意味モデル，意味規則などを習得しなければ適切に仕様を読み書きすることができず，使いこなすのにある程度の学習を要求される．この「形式体系/モデルの教育」が計算機科学の本質的な性格を反映した大きな問題であるとするDijkstraの主張と，それに関する興味深い論争に関しては[11]を参照されたい．

おわりに

形式仕様の現場での実践のためには，次のような要件を備えた形式仕様記述言語システムが開発されることが必要条件であろう．

・簡明な意味(モデル論的/公理的/操作的)を持つ．

・解析可能な仕様とそうでない仕様の境界線を示す理論がある．

・どのような仕様のどんな性質が解析可能であるかを示す理論がある．

・理論的に説明できない個別のknow howを示した例題のライブラリが整備されている．

・個別のknow howを体系化した方法(論)を整理した技法集が整備されている．

理論と実践の協調により，このような言語システムが(出来れば我国で)開発されることを期待したい．

参考文献

[1] 二木 厚吉: ISOにおける形式記述技法の標準化動向，情報処理，Vol.31，No.1，pp.3-10，1990.

[2] 大蒔和仁，二木厚吉: 形式仕様記述言語LOTOSの試用経験，情報処理，Vol.31，No.10，pp.1400-1413，1990.

[3] D.Bjorner, C.A.R.Hoare, and H.Langmaack (Eds.): VDM'90, VDM and Z -Formal Methods in Software Development, LNCS 428, Springer-Verlag,1990.

[4] K.Futatsugi: Trends in Formal Specification Methods based on Algebraic Specification Techniques -- from Abstract Data Types toSoftware Processes: A Personal Perspective --, Proceedings ofInfoJapan'90 Computer Conference, Part 1, pp.59-66, October 1990.

[5] J.Goguen and M.Moriconi: Formalization in programming environments, IEEE Computer, November 1987, pp.55-64.

[6] J.Guttag, J.Horning, and J.Wing: Some notes on putting formal specifications to productive use, Science of Computer Programming 2 (1982), pp.53-68.

[7] B.Meyer: On formalism in specifications, IEEE Software, January 1985, pp.6-26

[8] M.Moriconi (ed.): Proceedings of the International Workshop on Formal Methods in Software Development, ACM Software Engineering Notes, September 1990.

[9] J.Wing: A specifier's introduction to formal methods, IEEE Computer, Septermber 1990, pp.8-24.

[11] A debate on teaching computing science, CACM, December 1989, pp.1397-1414.

[12] IEEE Software, September 1990. (Featuring Formal Methods)

[13] IEEE Trans. on Software Engineering, Special Issue on Formal Methods in Software Engineering, September 1990.

パネル討論　理論は実践を導けるか，実践は理論を生かせるか？
— プログラミング言語設計における理論と実践 —

上田 和紀 ((財) 新世代コンピュータ技術開発機構)

もうかなり長い間，第5世代コンピュータプロジェクトの中で，プログラミング言語 (核言語) 関係の研究を行なっている．その技術的な詳細は参考文献 [1] ～ [4] にゆずって，本パネルでは，(核) 言語設計における理論と実践の関係について，経験に基づいてお話ししたい．

1. 言語設計と核言語

新たなプログラミング言語を設計し，それをコミュニティに定着させるのは，非常に困難で，しかも長い時間を要する作業である．強い存在理由と，継続的な努力がなければ，生き残ることは難しい．

ある応用システムを記述するとか，フォン・ノイマン計算機に非常に効率よくコンパイルできるとかいった目的で作った言語は，その設計基準や存在理由を外部に求めることもできよう．しかし，核言語の設計を開始した当初は，高並列コンピュータと知識情報処理を結ぶ，という (漠然とした) 目標があるのみであった．このような状況で，言語設計に何らかの必然性を持たせるためには，そのかなりの部分を言語自身の中に求めるほかない．

このような観点から我々は，核言語 (KL1 と，その基礎となっている GHC) の設計にあたって，次のふたつのことに留意した：

(1) 言語の根幹をなす原則，つまり設計指針や基本概念を明確にして，これを安易に変更しないようにした．例えば，「並列」と「並行」の両概念の明確な分離はその一例である [4]．また，副作用 (side effect) の排除という点でも，これまでに実装された非手続き型言語の中では非常に徹底している．これらの設計指針や基本概念は，新たな言語機能を考案，導入するときの判断基準としても有用である．

(2) (a) 処理系作成，(b) 記述力，(c) 理論的扱いの三者をバランスよく考えるように努めてきた．つまり，これらが言語を支え合うという形態を目指したつもりである．いままでに設計された言語を見ると，理論研究者のための版と実践プログラマのための版とができ，双方がほとんど遊離してしまっていることが多い．このような事態は，核言語ではできるだけ避けるように努めた．もちろん，(a) ～ (c) の各側面を最初から完全に配慮することは不可能である．しかし，これらのどの方向へも発展できるように配慮したつもりであり，また今のところは順調に発展しつつある．

2. 理論と実践の同時性

本稿で強調したいのは，言語設計にあたっては，言語機能と，(a) ～ (c) の三要素とを同時に考えることが非常に重要であるということである．言語設計にはパズルのようなところがあり，いろいろな角度から見た制約条件を同時に考慮してはじめて，よい解に到達することが多いようである．

この観点からいうと，プログラミング言語に関する理論研究の役割が，与えられた言語機能に理論的基礎を与えることにとどまってしまったらつまらない．言語設計の立場からは，ある言語機能が理論的観点から見てどうなのか，というフィードバックがほしいのである．理論的定式化を試みたら複雑怪奇なものができてしまったとしよう．その定式化自体はおそらく使いものにならない．このとき，定式化のしかたに問題はないかを考えるとともに，定式化の対象のほうに問題はないのかも考えてみてほしいわけである．

処理系作成にまで視野をひろげると，最近の高級言語と (通信の局所性が効率に大きく影響する) 低レベル計算機の組み合わせでは，処理系を効率化するために，かなり高度なプログラム解析をする必要がある．ここで，処理系を高速化するだけならば高度なプログラム解析の枠組を作ればよ

いのであるが，言語設計まで含めて考えると，このようなプログラム解析は，アドホックにしかできないよりは，(たとえば静的な型検査のように)系統的にできる方がはるかに望ましい．つまり，理論的定式化を，言語機能の形にして明示することができれば大きな前進である．

核言語の研究において，言語設計，理論，処理系作成の間のこのような相互交流が比較的うまくいった例をひとつ挙げよう．GHC では，プロセス間通信にユニフィケーションを使っている．ユニフィケーションのもつ情報の流れの双方向性は非常に強力であるが，並行言語の通信機能として使うときは，個々のユニフィケーションの断片で起きる情報の流れが静的に解析できたほうが，処理系作成の上で好都合であるし，それによって記述力が大きく犠牲になることもない．そこで，GHC プログラムの入出力モードの体系と，それに基づく静的解析の技法を提案した [2]．これはプログラム解析の手法の提案であると同時に，言語機能の提案にもなっている．また，文献 [2] で提案した処理系作成技法は，このモード体系によって初めて実現可能になった．このような相互交流は，理論と実践を分業でやっていたら容易ではないであろう．

3. (理論 vs. 実践) vs. (基本 vs. 発展)

高階の標題である．

以前から気になっていることのひとつに，「基礎」という言葉の使い方がある．この語は「理論」の意味で使うことが多いが，本当でそれでよいのだろうか？少なくとも，類語である「基本」についていえば，「基本的」であることと「理論的」であることとは独立であるように思えてならない．つまり，モノ作りの中にも非常に基本的な原則が見出される一方，理論の中に，将来何かの foundations として有効に機能しそうもないものも多い．

プログラミングの分野でも，相当数のシステムや論文が毎年生産される．その多くは，既存のものを発展させることに力を注いでいるが，既存の概念を整理したり，(abstract nonsense にならない程度に)一般化したりすることにももっと努力を傾けていいのではないかと思う．これは，理論か実践かにかかわらずである．この努力をしてゆかないと，計算機科学が科学として尊敬されるようにならないばかりでなく，計算機科学を学ぼうとするときに効率が悪くて仕方がない．計算機科学の根幹をなすような基本概念の数がそんなに沢山あるとは思えないのである．

自分の仕事の関連で言えば，GHC のような並行論理型言語の意味論の研究は近年，CCS, CSP, (非決定的) データフローのなどの並“行”処理の意味論と関連づける方向でかなり発展しつつある．プログラミングや計算のパラダイムやコミュニティをまたがって技術的な関係が生まれてくるのは気持ちが良い．一方，核言語関係の研究でこれまで立ち遅れていて，今後目指したいのは，並“列”計算の適切なモデル化を，並行／並列オブジェクト指向に興味を持つ多くの研究者と探ってゆくことである．

参考文献

[1] K. Ueda, Theory and Practice of Concurrent Systems: The Role of Kernel Language in the FGCS Project. In *Proc. Int. Conf. on FGCS'88*, ICOT, Tokyo, 1988, pp. 165–166.

[2] K. Ueda and M. Morita, A New Implementation Technique for Flat GHC. In *Proc. Seventh Int. Conf. on Logic Programming*, MIT Press, 1990, pp. 3–17.

[3] K. Ueda, Designing A Concurrent Programming Language. In *Proc. InfoJapan'90*, IPS Japan, 1990, pp. 87–94.

[4] K. Ueda and T. Chikayama, Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol. 33, No. 6 (Dec. 1990), pp. 494–500.

パネル討論：理論は実践を導けるか、実践は理論を生かせるか?

システムプログラミング
オペレーティングシステム・ネットワークの立場から

村井 純
慶應義塾大学環境情報学部

1　理論と実践

理論と実践は生物を主体とした根本的な概念である。自己を中心とした環境をとらえた結果が理論であり、それに基づきおこす行動が実践である。与えられた問いかけは、「理論は実践を導けるか、実践は理論を生かせるか?」である。実践を導かない理論は厳密には成立しない。どこかで、いつか、実践を導けない理論は存在し得ない。また、理論に基づかない実践は存在しない。その意味で問いかけに「理論を生かす」という言葉が使われているのなら、「理論を生かせない実践」というものは、すなわち狂気である。これを実践と呼ぶのは誤りである。

だからといって、理論が必ずしも即座に実践を導けるわけもなく、理論なしの実践もどきと実践の識別が困難であることも事実である。そこで問題は、理論を実践に導くために何ができ、理論を持たない狂気と実践をいかに正しく識別できるかという点である。もちろん、実践を即座に導けない理論が存在することを理由に、実践との関連を積極的に否定する理論が存在するとすればこれも狂気と呼ばなければならないし、また、健全な着想に基づかない、単なる実行を正当化する目的の理論も否定されなければならない。

2　システムプログラミング

コンピュータサイエンスにおける理論の背景は、人類の財産であるさまざまな理論体系と宇宙と社会、そして、コンピュータ環境である。コンピュータ環境そのものがこの分野の理論から導かれた実践の産物であるが、オペレーティングシステムやネットワークといういわゆるシステムプログラミングの分野の特徴は、この産物自身を環境の要素として強く意識した理論を出発点にしていることである。その意味でこの分野は悪い意味で実践性を重視してとらえられることが多い。これは、単純なモデルとしてとらえた対象となる環境とそれに基づき発展した理論が、大規模で複雑な実際の対象からかけはなれている場合が多く、必然的に理論と実践の結び付きが希薄になる現象がおこる。

しかし、このことが問題であるとは考えていない。このような関係からは実践、モデル、理論という相互のフィードバックによる一貫性をもった発展が実現するし、少

なくともそれが期待できるからである。実は、ここでの問題点の多くは、前に述べたように理論を持たない狂気と実践の混在である。正確に言うと、実践としての正当性を得るために作り上げた理論もどきをともなっているのがここでいう狂気である。システムプログラミングにおけるこのような現象は、政治、商売、流行という要素によって引き起こされる場合が多い。結局、コンピュータサイエンスの分野全体の理論と実践の健康な関係はこの3要素に蝕まれているのである。

3 理論と実践の正しい関係のために

理論と実践によるコンピュータサイエンスの発展に携わる研究者の仕事は、この両者の正しい関係に基づかなければならないことはいうまでもない。実践に結び付き得る理論と、新しい着想を導き、理論の発展を促す環境を作りだし得る実践は、いずれも一貫性のある思想、哲学、原理に基づいていることが最も重要である。これらをそれぞれ中心的な役割と自負する研究者は、それぞれ、理論派と実践派と呼ばれることになるが、上記3要素に基づいたいわば「情報派」としての役割を果たさないことが教訓である。

参考文献

[1] Werner Heisenberg DER TEIL UND DAS-GANZE Gesprache im Umkreis der Atom-physik 湯川訳 みすず書房, 1974.

パネル討論: 理論は実践を導けるか，実践は理論を生かせるか？
プログラミングの理論と実践における自律と共生
Autonomy and Symbiosis of Programming Theory and Practices

安村 通晃　慶應義塾大学 環境情報学部*

1　はじめに

過去 20 年間のチューリング賞の授賞者の研究内容をみてみると実に過半数の 11 件がプログラミング言語とそのシステムに関するものである [1]．プログラミングは，これまでもそして，これからもコンピュータサイエンスにおける主要な研究分野であり続けるであろう．

プログラミングに関する研究アプローチとして，他の研究分野と同様に，理論的な研究と実践的な研究とがある．この二つのアプローチは，お互いにどのように関連しあって発展しあっているのだろうか？その答として定まったものはまだないが，ここでは，プログラミングの理論と実践における相互作用を，それらの自律性と共生という関係で捉えてみたい．

2　プログラミングにおける理論と実践の自律 ・共生モデル

(1) 理論が先か，実践が先か？

自然科学の研究においては，自然現象があり，それらを人間が日々経験している．その経験の中から法則性を見つけだすのが理論であるので，実践（実験）との関係では，初期の理論研究が先行することは少なくない．

ところが，コンピュータに関する科学（コンピュータサイエンス）では，まずは，実際のモノづくり＝プログラミングを行う必要がある．したがって，まず，最初に実践があり，実践の研究の過程で，理論的な研究の芽（問題意識）が芽生えるのである．

*Michiaki Yasumura　KEIO University, Faculty of Environmental Information

(2) 理論と実践の自律性

いったん，理論的な研究が始まると，実践と理論はそれぞれ，独立した道を歩み始める．理論的な研究の問題意識と実践的な研究のそれとが，決定的に食い違うからである．理論的な研究においては，いかに問題をきれいに定式化し，それから形式的な意味での新しい結果がいかに引き出せるかが，課題となる．一方の実践は，具体的に役に立つプログラムをいかに作るかの方に専念せざるを得ない．理論は空を駆け，実践が地を這い始めるのである．

(3) 理論と実践の共生

しばらく，別個に発展していた，理論と実践の研究はともに行き詰まり始める．理論の方は，新しい結果が出なくなったり，人々の関心が薄れたりするからである．一方の実践は，最初のうちは力づくでプログラムを作り上げ，そこそこの役に立つものができるが，これをさらに改良を加えたり，新しい人たちに引き継ごうとしたときに，自分達のやってきたことを整理する必要に迫られるからである．こうして，理論は実践からその生き生きした息吹を受け取り，実践は理論から暗闇の中の光明を求める必要を感じる．共生の開始である．

3 事例研究

次にプログラミングにおけるいくつかの研究分野で上で述べたモデルは本当に正しいのであろうか？仮説の検証を試みよう．

(1) コンパイラの理論と実践

コンピュータサイエンスのもっとも初期の貢献は，おそらく高級言語の開発とそれと一体となったコンパイラの開発であろう．この場合は，明らかに実践（コンパイラの開発）が先にあり，その後のコンパイラの理論的な研究の引き金になっている．しかし，関連する理論研究としての文法理論やオートマトン理論は，必ずしもコンパイラの研究に（影響は与えられているが）直接起動をかけられた訳ではない．これは，文法理論については，自然言語のほうの研究もあったし，また，オートマトンは，コンピューター一般の研究が背景にあったからである．コンパイラの理論と実践における自律と共生については，いくつかの例をあげることができる．(最近のいい例は，最適化コンパイラの研究であろう)

(2) ロジックプログラミングの理論と実践

この場合は，実践よりも理論が先行しているように見える．これは，この研究分野がコンピュータサイエンス単独で発達してものではなく，論理学という異分野から影響を受けて発展したためである．この場合も，resolution という理論的な研究が実践の研究に刺激を与え，さらにPrologの言語と処理系の開発という実践的な研究がその後の理論的な研究を活性化するという共生関係が見られる [2]．

(3) オブジェクト指向プログラミングの理論と実践

オブジェクト指向の研究の先駆けが Smalltalk であることは，だれも異存が無いであろうが，それが Actor をもとにしたかどうか，定かではない．むしろ，Simula67 を参考にしたとも考えられる．Smalltalk は，オブジェクト指向の実践的な研究であるが，理論的な研究としては，先に述べた Actor の他に，型や継承の研究がある．特に後者は，実践的な研究からいい意味での影響を受けて発展するだろう．

4 おわりに

どうやら，実践から理論が生まれるという，単純な関係だけでは無いことが分かったが，いずれにせよ，理論と実践の自律性を尊重しつつ，互いに共生関係にもっていくことが，今後も大切であろう．

参考文献

[1] ACM Turing Award Lectures - The First Twenty Years, 邦訳，ACM チューリング賞講演集，共立出版.

[2] 溝口文雄著，人工知能の研究者たち，共立出版.