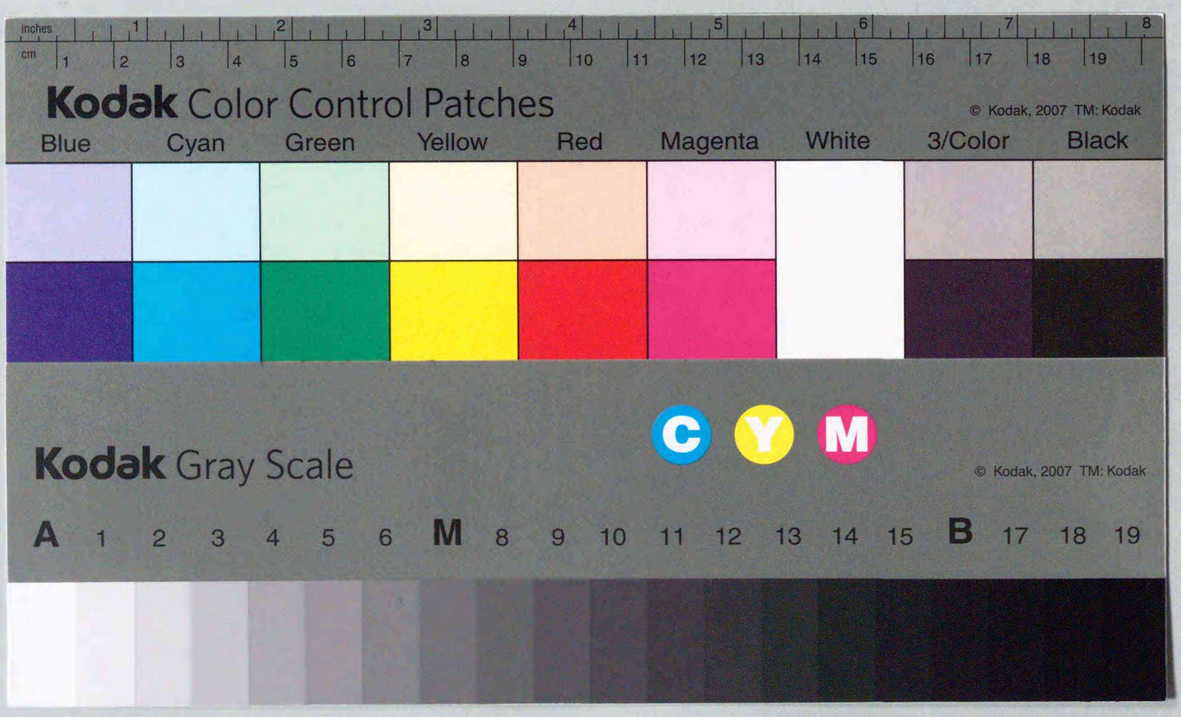
ソフトウェアの形式的仕様化と検証に関する基礎的研究

林 慶一



ソフトウエアの形式的洗練化と検証に関する基礎的研究

Fundamental Studies on Formal Refinement and Verification for Software

林　雄二

平成１２年２月４日

目 次

1 はじめに

本論文はシステムの仕様および設計における表現法として IO 正則式を提案し、その表現の下で仕様の洗練化，プログラムに対する検証の方法，さらにデータフローネットワークの検証への適用法について報告するものである。

一般に，プログラムはデータ操作と制御の両面を表現している。入出力データ構造に基づく分析をするジャクソン法 (JSP)[21] では、制御とデータ操作を分離して表現する。ジャクソン構造図における最下層の箱はデータ操作であり、構造図は正則表現に類似したものである。構造化プログラミングによる基本３構造も，巨視的には正則構造にほかならない。プログラムにおいては，データ操作に実行順序の制御が加えられてアルゴリズムとして組み立てられている。

アルゴリズムには、論理によって表現できる部分、すなわち実行順序に左右されない部分と実行順序が意味をもつ部分の両方があり、手続きプログラムでは、両方を区別して表現することはできない。一方，関数型，論理型の宣言的プログラムは静的な関係によって問題を表現するもので、アルゴリズムの動的性の部分を表現することは困難であり、そのために，現実的な広範囲の問題にまで宣言的プログラミングが適用されるまでには至っていない。プログラムの静的関係の部分と，動的な振る舞いの部分を区別して表現することは、問題の分析，設計段階に有効である。本論文で導入した IO 正則式の基本単位は、評価順序が意味を持たないデータ間の静的な関係式（論理式）であり、正則表現によって、評価順序が意味を持つ動的な振る舞いの部分を表現することになる。

近年，形式的技法（Formal Methods) が研究され、なかでも形式的に仕様記述を行うための言語Ｚ [33, 39, 8]，VDM[28]，Ｂなどが脚光を浴びている。特に、Ｚは現実的な問題に適用され，その効果が評価されている。Ｚはスキーマによってプログラムの静的な論理関係部分を表現すると共に、操作の事前状態，事後状態を表現することによって状態の変化を記述する。このようなスキーマ表現によって、問題が形式的に表されると、形式的記述が仕様を満たしているか否かの検証（Verification），仕様に過不足や矛盾が無いか否かの妥当性確認 (validation) を効果的に行うことができる。現在形式的技法が利

用されている多くは，この検証と妥当性確認の部分である。

Zのような形式的仕様記述には，システムの動的性を記述する部分が不足している。複数のスキーマで表された仕様に対し、スキーマの実行順序関係は直接的には表現されていない。実行順序を表すには、スキーマが持つ変数に状態を保持させて、個々の状態において、どのスキーマが実行されるかを表現する必要がある。しかも、Zの仕様に基づいて導かれたプログラムは、一つのcase文で，実行されるスキーマを選択するような形式のものになる[33]。そこには、反復計算されるスキーマ，最後にだけ実行されるべきスキーマなどの区別が表面的にはなく、毎度case文が状態変数やイベントを判断して、スキーマの選択を行う方式のプログラムに成らざるを得ない。これは、構造化プログラミングと相いれない考え方である。スキーマを基本項とするIO正則式によって仕様を表現しておくことは、スキーマの実行順序，反復，選択などの構造が表現されるので、この問題に対する一つの解決法になる。特に、環境との相互作用のある問題に対しては、仕様に環境からの入力や環境への出力の実行順序を盛り込んでおく必要がある。入出力を伴う問題の仕様を，論理型，関数型などの宣言的プログラムで記述することが途方もなく困難であることは知られている[43]。同様に、Zのスキーマがシーケンス（あるいは配列やキュー）などによって、入出力データを静的なデータ構造として表現しても，外部との相互作用を表現することは困難である。IO正則式によって、相互作用が順次進行する過程を記述することが可能である。

基本的にZは静的な表現による仕様記述であり，システムの振る舞いを表現することには十分対応していない。そのために，いくつかの提案がなされている。それらは，Zスキーマを包含した動的性を表現する仕様記述と，Zスキーマ中に動的性を盛り込む仕様記述の，大きく２通りの方法に分類することができる。筆者のIO正則式による仕様記述は前者である[51]。前者としてそのほか，CSP[18],CCS[30]などとの結合を試みた仕様記述の研究[13, 12, 34, 37]がある。これらは並行プロセスを含むモデルを対象にしており，手続きプログラムへの洗練化を目的とするときには，必ずしも適しているとはいい難い。また，環境との相互作用があるシステムの振る舞いを状態遷移によって表現する方法の研究[35]

もある。これは形式化が困難であり，状態遷移と構造化された目的プログラムとの関係付けが困難である。後者に分類される方法として，操作や状態のヒストリをZスキーマに盛り込む方法[9, 11]，時制論理の記述をスキーマの述語部に含めた表現[10]なども提案されている。いずれも，入出力を伴う個々のプログラムとの対応が明らかではなく，スキーマに新たに振る舞いを盛り込むことは洗練化や検証における複雑さを増すことになる。

通常のZ仕様では、個々のスキーマはプログラムの各部分に対応し、それら部分の統合を直接的に表現するスキーマは一般に存在しない。１つのプログラムに複数のスキーマを羅列したものが対応していることが、プログラムの正当性検証を困難なものにしている。IO正則式では、プログラムの正当性を元のIO正則式の仕様に照らして検証することも可能になる。IO正則式がプログラムの構造に対応するからである。筆者は、IO正則式の仕様に対し、ホーア論理[17, 45]の基に、プログラムの正当性検証を行う方法[51]を提案する(第５節)。

Zなどの形式的仕様を洗練化によって、段階的にプログラムに変換しようという研究も進められている。これは、洗練化計算(Refinement Calculus)[1, 39, 31]として知られる研究分野である。現在の洗練化計算は，Zのような形式的仕様から制御構造を見いだし、最後はプログラム（あるいは、ダイクストラの護衛コマンドプログラム[7]）に変換しようというものである。ダイクストラは，彼の著書[7]で、プログラムを証明をしつつ段階的に洗練化していく方法を提案しており、現在の洗練化計算は彼の考えに基づいている。洗練化計算の表現は、論理式で表された仕様とプログラムの制御文（while文,if文など）とを組み合わせた表現(specification statemente)を用いた記述法を基にしている。 これは、洗練化においては、仕様段階とは異なった表現法が必要になることを意味している。ソフトウエア開発において、多くの方法論が実践化されずに終わるのは、それら方法論の記述方法が，開発のそれぞれの段階で異なっている点にも原因があるとの指摘もされている。例えば、構造化分析[6]における仕様記述ツール（データフローダイアグラム）などは、構造化設計[6]の段階には全く利用されず、代わりにモジュール構造図などの異なるツールが必要とされる。IO正則式で表現することは、仕様から設計まで、統一した

記述方法を用いて開発を進めることを意味する。そのことで、各段階で単一のスキーマを変換するのみならず、スキーマの合成に対しての変換を進めることが可能である。また、従来の洗練化の過程では、プログラムに必要な制御条件（while 文の判定条件など）を順次開発していく必要があるのに対し、IO 正則式では、それら条件を切り出すことは最終のプログラミング段階にまかせることになる。このことはちょうど、ジャクソン法が、ジャクソン構造図の段階には、プログラムの制御文 (while や if-then-else など) の条件式に腐心する必要がなく、すべてプログラミングの段階に任せていることと符号する。筆者は、IO 正則式を用いた洗練化手法を提案する (第 4 節)。

データフローネットワーク [22, 25] は複数のプロセスをチャネルで結んだ計算モデルである。個々のプロセスは入力データストリームを出力データストリームに変換するプログラムである。プロセスの機能を外部から見たとき、それは、ストリームからストリームへの関数とみなすことができる。その考えから、関数プログラミングや論理プログラミングで、ストリーム [43] をデータとして扱うことが提案されてきている。しかし、これら宣言的プログラムで環境との相互作用（すなわち入出力）を表現することは，データフローネットワークに比して著しく困難である。データフローネットワークという計算モデルに対して、基礎になっている性質は不動点との関係である。すなわち、プロセスを関数とみた場合に、ネットワークに生ずるストリームは関数の不動点であるという性質である。筆者は，数学的なこのような性質が、具体的にプログラム実行の下で、どのような性質に該当するか、すなわち、有限で計算が終了する場合、ストリームが通信チャネルに残って終了した場合、などに対する不動点の意味を考察した [48](第 6 節)。データフローネットワークにおいて、個々の入力データにプロセスがどう反応するかを盛り込む立場では、順次受け取る入力データに伴ってどのように状態を変え、どのような出力を送出するかを表現しなければならない。そのような表現として、IO 正則式は効果的である。データフローネットワークの個々のプロセスを IO 正則式で表現することにより、ネットワーク全体の性質の検証をネットワーク内を流れるストリームの不動点の性質 [48] を利用して行うことが可能となり [50]、筆者はそのための手法を提案している (第 7 節)。

以下では、第 2 節で、ソフトウエア開発方法論の現状をふまえ、IO 正則式の意義を考察する。第 3 節では、Z と洗練化についての現在知られている方法を解説する。さらに第 4 節では、Z スキーマ上の IO 正則式による洗練化の手法を提案し、第 5 節では、プログラムの正当性を IO 正則式の仕様を基にして検証する方法を提案する。第 6 節では、データフローネットワークにおける不動点の持つ意味を考察し、第 7 節では、データフローネットワークにおける仕様の不動点を用いた検証に，IO 正則式を用いる方法の提案をする。

2　ソフトウエア開発方法論の課題

2.1　プログラムとアルゴリズムの分析

多くの場合，アルゴリズムは発見的に見いだされなければならない。問題をアルゴリズムに基づいて分類することが可能であれば、問題によって、見いだすべきアルゴリズムのパターンが得られ、初心者に対してもアルゴリズムの開発を容易にすることが期待できる。オブジェクト指向において、昨今デザインパターンが唱えられている。これは、オブジェクト指向による設計にパターンを利用しようというものであり、問題の設計を分類されたパターンに基づいて行うことが可能となる。オブジェクト指向には，継承，インスタンス化，関連，集約など、多くの機構があり、それらの組み合わせにより様々なオブジェクト指向プログラミングが可能になる。デザインパターンは問題にパターンを導入し、設計をいわば再利用しようというものである。それに対し、アルゴリズムに関しては、具体的な分類の指針があまり得られていない。計算処理の対象分野として、ファイル処理分野，数値計算分野，リアルタイム処理分野等の分類はあるが、さらに詳細な分類はなされていない。一方，データ構造や抽象データ型などによって、データを基にアルゴリズムを組み立てる方法が提唱されている。Wirth [38] は、プログラムを以下の要素に分析している。

プログラム＝アルゴリズム＋データ構造

代表的なデータ構造として，例えば以下のようなものが上げられる。

データ構造 { 配列スタック / 行列 / キュー / グラフ / ネットワーク

データ構造にアルゴリズムが依存し、一般に、問題に適したデータ構造を利用することで、アルゴリズムはより単純化される。しかし，データ構造によって、アルゴリズムが機械的に定められるものではない。一方ジャクソン法（JSP）[21] は、入出力データ構造に基づいてプログラム構造を作ろうという設計法であり、アルゴリズムを定まった手順に基づいて導出することを狙いとしている。JSP により，入出力データ構造から、プログラムの制御構造の基本形が得られ，詳細なプログラム文は構造図に最後に割り付けられる。これは、優れた設計法ではあるが、入出力データ構造に，対応関係が無い場合（構造不一致 [21]）には、特殊な操作をしなければならず、また、それによって得られるプログラムは決して期待されるような分かりやすいものではない。JSP の立場では、問題を構造一致と構造不一致の２つに分類することができ、構造一致に対しては、プログラム構造を入出力データの仕様から導くことができる。

ジャクソン法 (JSP) { 構造一致 / 構造不一致

有限状態機械を基にした構造一致問題に対しての形式化は Hughes [20] によってなされており、Hughes は有限状態機械をプロセスとすることで、構造一致のケースをモデル化した。すなわち、プログラムをあるレベルの粒度の下で分析した場合に，構造一致であるとは、その粒度の下での入力出力の対応が有限状態機械で処理できることであるとの結果である。そのような多入力多出力の有限状態機械の性質は筆者によって研究されている [47]。さらに構造不一致問題に対する形式化と分類は、筆者 [46] によって研究された。有限状態機械と反転機構を直列またはフィードバック状に結合した機構であらゆるアルゴリズムが実現されることを示した上で、バックトラック型や（有限的）順序不一致型などの問題を、基本の機構のどのような結合によって表現できるかを，形式的に表現することがなされた。しかし、直接その類型化からプログラムに結びつける試みはなされていなく、他の構造不一致に対しては、形式化するに至っていない。このようなことから、構造不一致の問題こそがアルゴリズムへの洗練化を研究すべき対象であるといえよう。

ジャクソン法で用いられるジャクソン構造図は、正規表現に表すことができる。その場合、正規表現の基本要素は、データ操作の名前である。敢えて記述すれば、構造一致の場合は，基本的に入出力データの関係を表現する関係式になる。このような正規表現が

与えられれば、プログラム構造は基本３制御構造で置き換えることができ、入出力データの関係式を単独のデータ操作文または複数の文からなるルーチンに変換することができる。

1. 基本３制御構造　...連接，選択，反復
2. データ操作 ...代入文，入出力文，関数 (手続き) 呼び出し文 ...

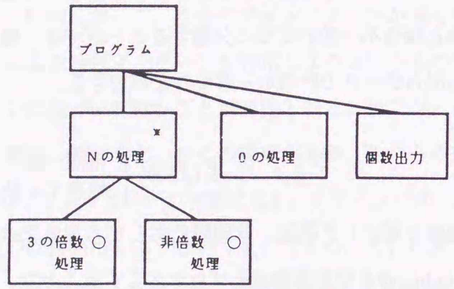


図 1: ジャクソン構造図

IO 正則式は、(入出力) データの関係式を基本要素とする正規表現である。そこでは、プログラムの制御部とデータ操作部を分離して表現することになる。IO 正則式の詳細は、後述されるが、簡単な例を示しておくことにする。例えば、多数の製品の重量を入力し (ただし入力データの終了は重量 ≤ 0 で表す)、100 未満のときのみ、不良という文字列を表示するアルゴリズムは，以下の IO 正則式で表現される。

$$[\,[weight? \geq 100] \sqcup [0 < weight? < 100 \wedge out! = \text{"不良"}]\,]^*[weight? \leq 0]$$

後の節で我々は、基本要素をＺのスキーマとした IO 正則式の表現も用いることになる。

Prolog の開発に大きな影響を与えた Kowalski[27] によるアルゴリズムの捉えかたは、

アルゴリズム＝論理＋制御

で表現される。この場合の制御は論理式を評価する順序を意味しており、論理式の羅列は静的なデータの関係を表現するに過ぎないが、評価の順序が加えられると、その評価

順序に依存して異なる実行過程が得られるというものである。しかし彼の制御は、プログラムの実行順序制御のような複雑な制御を意味するわけでなく、状態の変化を表現するものでもない。すなわち、宣言的プログラムにおける表現と再帰などの評価の両面を意味している。アルゴリズムには、動的に状態を変化させるという意味があることをふまえて、筆者は IO 正則式によって、

アルゴリズム＝データ操作＋制御

であるという立場をとる。すなわち、論理ではデータの関係を表現するのに対し、データ操作では、状態の（事前と事後の）変化を表現することになる。前者 (Kowalski) に対し、後者はよりプログラムの直接的意味に近い分析である。Ｚ [33, 39, 8], ＶＤＭ [28] などの形式仕様記述言語は状態の変化を記述する形式，すなわちデータ操作を表現するが、制御を直接表現はしない。その反面、Ｚのスキーマなどには、データ構造（すなわち型）を表現する機能は十分に備わっている。したがって、Ｚについては、

スキーマ＝データ構造＋データ操作

を表現している。

IO 正則式の基本要素をＺのスキーマにすることによって、IO 正則式の持つ意味は、「Ｚスキーマ＋制御」で表される。従って、

	IO 正則式	
＝	Ｚスキーマ＋制御	...Ｚ上の *IO* 正則式より
＝	データ操作＋データ構造＋制御	...*Z* の性質より
＝	アルゴリズム　＋データ構造	...林の見解より
＝	プログラム	...*Wirth* より

なる関係が得られる。このことは、IO 正則式によって、開発すべきプログラムの仕様，設計を表現することが可能であることを意味している。

2.2 ソフトウエア開発方法論

ソフトウエア開発における分析，設計の技法は以下のような変遷を経て現在に至っている [52]。1970 年以前は、職人芸的プログラミングの時代であり、プログラムを開発することに精いっぱいで、方法論と言えるものがほとんど開発されていない時代である。1970 年代にはいると、プログラミングの技法と，主に事務計算分野での方法論が普及し始めた。1980 年代には、リアルタイム分野に対する方法論が普及し、オブジェクト指向プログラミングが生まれたが、この時代には広く普及するには至らなかった。1990 年代には、オブジェクト指向分析, 設計が取り入れられ、オブジェクト指向に対する多くのツール, 方法論が展開されてきた。さらに、形式的技法が生まれ、少しずつ現実的なシステム開発に適用されるようになりつつある。

1. 1970 年代
   - 構造化プログラミング
   - ジャクソン法 (JSP)
   - ワーニエ法
   - 構造化分析設計
   - データ抽象化法
2. 1980 年代
   - リアルタイム構造化分析
   - オブジェクト指向プログラミング
   - ジャクソンシステム開発法 (JSD)
3. 1990 年代
   - オブジェクト指向分析, 設計
   - 形式的技法

オブジェクト指向がそうであったように、ソフトウエア開発現場には慣性が働いており、新しい方法論が本格的に普及するには、相当の期間を必要とする。形式的技法についても、早急にその普及が拡大するとは限らないが，おそらく、次の時代には一層の浸透が進むものと思われる。

このように、従来から適用されてきている構造化分析, 設計と近年普及し始めているオブジェクト指向分析，設計、さらに今後の普及が期待される形式的技法（Z，VDM など）の３つが，現在の代表的なソフトウエア分析設計の指針であるといえよう。なお、形式的技法は，オブジェクト指向に対するＺ [10] などが提案され、オブジェクト指向のもとで適用することも可能になってきている。

ソフトウエア分析設計の基本的指針
- 構造化分析, 設計
- オブジェクト指向分析, 設計
- 形式的技法

構造化分析 [6] は問題をデータフローに基づいて分析し、プロセスとデータフローによるデータフロー図（ＤＦＤ:dataflow diagram）にモデル化する。その後、構造化設計によりモジュールに分割し、モジュール構造図に表し、個々のモジュールを設計する。ＤＦＤにおけるプロセスは、入力データのストリームを出力データのストリームに変換する機構であり、IO 正則式で表現することが可能である。オブジェクト指向では、個々の存在がオブジェクトによって表現される。オブジェクトはクラスによって表され、クラスの状態変化は、クラス内のメソッドによって実行される。メソッドがどのような条件下で、どのような順序で実行されるかは，明示されるべきクラスの性質の一つである。IO 正則式はこのようなクラスの振る舞いをも表現することが可能である。また、現在のプログラミング言語においては、そのようなオブジェクトの振る舞いを表現する手段は持たないので、分析, 設計時に必要とされる表現である。

2.3 ソフトウエアに対する形式化

ソフトウエアの開発において、要求分析，システム設計，プログラム設計，プログラミングと，順次進行させる開発プロセスモデルを、落水モデル (waterfall model) という。

しかし現実の開発において、仕様の不備、仕様の変更に伴う後戻りが頻繁に生ずるので、落水モデルを忠実に適用することは、一般に賢明とはいえない。このことは、ソフトウエア開発が抱える困難の一つである。ソフトウエア開発方法論が目標とするのは、このような困難を克服すべく、プログラムの機械的な生成 (自動的な生成あるいは合理的な手順に沿った生成) であるといっても過言ではない。そのためには、システムの仕様が厳密に表現されなければならない。形式的仕様記述 (formal specification)[45] はその目的の為に生まれてきた表記法である。形式的とは，すなわち数学的であることを意味する言葉である。仕様を形式化することができれば、洗練化によって順次、プログラムに近づけることが可能になる。厳密な規則に基づいて洗練化を進める方法は、ダイクストラが提唱しているものである。一方、順次洗練化するよりも、開発されたプログラムの正当性を証明する方法も、形式的仕様によってさらに現実化されることが期待される。この場合は、ホーア論理 [17] など、プログラム証明の規則が利用されることになる。

このような形式的仕様記述は，今後次第に普及を広めるものと思われる。そのときは、分析者，設計者に数学的素養が要求されるが、翻って他の工学分野の設計者を考えてみると、機械工学, 電子工学, 建築工学など、いずれにおいてもそれぞれの分野での数学的素養を必要としているのである。ソフトウェアシステム開発に数学的素養が不必要でよいわけがあろうか。

Z,VDM などの形式的仕様記述では、制御までを仕様に直接盛り込むことはできない。状態を保持する変数を用意し、変数の値をスキーマ実行の条件とすることで、実行制御を表すことになる。それは、仕様に内部の状態変化を表す余分な変数を加えることになり、望ましいことではない。入出力処理が中心のシステムなど、環境との相互作用のある問題では、システムの状態変化は環境側に依存することになる。すなわち、制御を仕様に盛り込むべきものであり、ＩＯ正則式は、そのようなシステムに対する仕様表現として有効である。ささらに、洗練化によって、プログラムに接近する場合の表記として、仕様段階と同じ表記を用いることができる点も優れている。問題によって、IO 正則式が要求仕様であることも、設計表現であることも、どちらの場合もありうる。

IO 正則式が制御とデータ操作の分離したシステムの記述であることは、要求仕様と設計の両面を備えているので、洗練化の過程をもこの表現ですすめることができることを意味する。そこで、従来からの洗練化とは異なる表現法、異なる規則による導出が可能である。

計算機科学は、数学とも異なる科学である。それはコンピュータという、人類が開発した計算機構でありながら、人類が初めて出会うほどの多様な可能性を秘めたものを対象とする。コンピュータは自らの状態を変化させながら計算を進める機構であり、その動的性は静的な世界を扱う従来からの数学では表現することがなかった部分であるということができよう。そのために，計算機科学は独自の論理を模索することになる。そのようにして生まれるものの多くは数学の分野として包含されていくであろうが、コンピュータを契機として考え出されたものであることは，コンピュータの可能性の深さを示すものである。そのような計算機科学の論理として、

1. 時制論理
2. プロセス代数
3. 動的論理，ホーア論理
4. 正規表現, オートマトン

などを上げることができる。IO 正則式は，その形式は正規表現であり、その基本要素は数学の分野としての論理式である。コンピュータに関するすべての問題を，計算機科学独自の理論で覆い尽くすことは不可能であり、多くは従来の数学に還元して究めることが必要になる。

2.4 システムの分類と抽象化

コンピュータを基にしたソフトウエアシステムは，その性質により、以下のように分類することができ、また、これら性質をもったプログラム単位の集合とみなすことができる。以下にそれぞれのシステムに対応する主たる形式化の手法と思われるものを列挙

する。

1. 並行処理
   (a) 同期方式 ... CSP,CCS 等
   (b) 非同期方式 ... データフローネットワーク

2. 逐次処理
   (a) 入出力処理主体 ... データフローネットワーク,JSP
   (b) 内部状態変化主体 ... Z,VDM

並行処理プログラムのなかで、同期方式のプロセス間通信によって表現することができる問題は、CSP[18],CCS[30] 等によって仕様を表現し分析することが提案されている。非同期方式に対しては、プロセス間のチャネルが待ち行列となり、プロセスはストリームを入力しストリームを出力するモデルとして，データフローネットワーク [22] により表現される。本論文が主たる対象とするのは、逐次処理プログラムであり、逐次処理プログラムは、入出力データ処理を中心にするものと内部状態変化を主として実行するものとに分けられる。入出力データ処理主体のものは、データフローネットワークにより分析され、さらに JSP へと設計を進めることが可能になる。一方入出力処理が中心ではないシステムは、データフローによる分析を適用することが必ずしも容易ではない。この場合は、内部データ構造を設計し、アルゴリズムへと洗練化していく方式が必要になり、Z,VDM などのようなデータと操作の両面にわたる表記法が有効である。データフローネットワークは、各プロセスがストリームをストリームへ変換する関数とみなすことができるので、半順序集合上のストリーム関数としてプロセスをとらえることができる。その場合、関数の不動点がこのネットワーク上に現れるストリームであるという性質があり [22]、その性質を用いてデータフローネットワークの性質を検証する事が可能になる。数学上での不動点が関数を基にしているのに対し、現実のプログラムは、入力文、出力文の実行によって、引数値，関数値を意味することになる。筆者 [48] は現実のプログラムにおける不動点の意味，例えば入力されないデータのある場合，無限に計算

が進行する場合等に対する不動点の意味を考察している。

3　仕様記述言語Ｚと洗練化

3.1　形式的仕様記述

形式的技法 (formal methods) は、ソフトウエアの開発に，数学に基づいた表記法を用いることによって，システムの性質を表現し，厳密な方法で開発を進めようという手法である。従って形式的技法は、仕様化，設計，さらに検証に対し、組織的な方法で取り組む方法も提供する。

形式的技法は、システムに内在する曖昧さ、不完全さあるいは矛盾を解きあかす。システム開発の初期段階で使われれば、プログラミングやテストなどの開発の下流工程で誤りが見いだされ、開発工程を後戻りしなければならないことを避けることができる。もし形式的技法が下流工程で用いられると、実現されたソフトウエアが正しいものか否かの検証を行うことに効果を発揮する。プログラムの正当性検証，証明に基づく仕様の段階的洗練化などは形式的技法に包含される手法であるが、中心になるのは、形式的仕様記述言語である。形式的仕様記述言語は、開発されるべきシステムの機能の必要十分な条件を形式的（すなわち数学的に）表現する言語である。Ｚ，ＶＤＭ，Ｂ，ＣＳＰ，ＣＣＳ，ＬＯＴＯＳなどが知られている。

自然言語を用いた仕様記述が正確性を欠くことはいうに及ばない。仕様の不完全さが、その後の設計，プログラミングの段階に多大な影響を与えていることは、しばしば指摘されてきている。しかし、プログラミング言語と仕様記述とを同一の表現にすることは意味がない。仕様段階では、何を作るか (what) が目的であり、設計のプログラミング段階ではアルゴリズムも含めて，どう作るか (how) が目的である。プログラミング言語は前者ではなく、後者のために用意されている言語である。仕様を厳密に表現するには、形式的記法、すなわち集合，記号論理，関数などに基づく数学的記法が必要である。意味が明白になると共に、数学の証明技法が適用出来，数式に対し数学の演繹的導出を適用することも可能になる。さらに、定まった構文に沿った表現にされることで、コンピュータによる演算操作によって仕様を加工することも可能になってくる。

Brooks は彼の著書 [4] で、「銀の弾丸は存在しない」という言葉を表明している。これ

は、ソフトウェア開発に伴う様々な困難を，すべて解決することができるような理想的な技術は存在しないことを唱ったものである。その中で彼は、今まで銀の弾丸になり得ていないいくつかの技術のうちの一つにプログラムの正当性検証を挙げている。労力の大きさ、検証自体に誤りが含まれる可能性, などをあげ、むしろ完全で矛盾のない仕様を作ることがソフトウェア開発の最大の問題であると指摘している。Ｚのような仕様記述言語が生まれた今、仕様を表現するための優れた手段が得られたことになる。この先、仕様記述言語の下での理論の発展や，証明, 洗練化を助けるＣＡＳＥツールの開発が，形式的技法を銀の弾丸へ近づけることになるかもしれない。

3.2　Ｚの概要

Ｚ [33, 39, 8] の仕様はスキーマ (schema) というシステムの状態を記述するフォームの集まりである。それぞれのスキーマでは、事前の状態がこのスキーマ評価後にどのような事後状態に変化するかを記述する。スキーマ内には１階述語論理式が含まれるが、事前変数と事後変数を含めることができるのが、通常の論理式と異なる点であり、Ｚの特徴的な点である。また、Ｚのスキーマでは変数の型宣言を含み、関数型言語が持つような豊富な型を用いることができる。この型によって型チェック (type-checking) を行うことで、仕様に矛盾が生じていないことをチェックする助けになる。

現在、Ｚは主に仕様の検証，妥当性確認において実践的に利用されている [39]。すなわち厳密に表現された仕様をもとに、その仕様が要求としての性質を盛り込んでいるかを検証 (verification) し、また，仕様の中に矛盾が含まれていないかの妥当性を確認 (validation) することに利用されている。研究面では、洗練化 (refinement) の試みが進められている [39, 31, 1]。Ｚの仕様に対し，従来からの洗練化手法 (洗練化計算 [31]) を適用することが原理的には可能である。洗練化は規則に基づいてスキーマを変換していく過程であるが、直接的にプログラムを開発して正当性を証明することがむしろ現実的な場合もある。その為には、Ｚ仕様記述とプログラムとが構造的に対応していることが必要になる。

以下では後の説明に必要な部分に限定して、Ｚの簡単な紹介をする。Ｚのスキーマ

(schema) は以下の形式から成る。

$$
\begin{array}{|l}
\hline
< schema\ \ name > \\
< declaration > \\
\hline
< predicate > \\
\hline
\end{array}
$$

これを水平的記法で [$< declaration > | < predicate >$] と記述することもある。

Example 3.1 ある不動産屋で、売り家の集合 *(Houses)* があり、そのうち幾軒かが何人かの客 *(Customers)* に販売 *(sold)* された。この関係をZのスキーマで表現する。

$$
\begin{array}{|l}
\hline
Estate \\
Houses : \mathbb{P}\, AllHouses \\
sold : AllHouses \pfun Customers \\
\hline
dom\ sold \subseteq Houses \\
\hline
\end{array}
$$

スキーマ名 *(Estate)* は、不動産屋システムを記述している本スキーマに付けた名称である。売り家の集合 *(Houses)* は，家全体 *(AllHouses)* の部分集合である。すなわち、*Houses* の型は、*AllHouses* のべき集合となる。販売実績 *sold* はどの家がどの客に売れたかを表す部分関数である。すなわち *sold* の型は *AllHouses* から *Customers* への部分関数になる。述語部では、販売実績の *domain*（すなわち購入された家全体）は、*Houses* の部分集合であることを意味している。

宣言部，述語部共に各行は連言 (conjunction) として結合されていると解釈される。水平的に記述する場合には、セミコロン (;) によって区切って記述すればよい。このスキーマはシステム全体を通して不変に保たれる性質を記述しているスキーマである。次の例で、スキーマ実行の前後における状態の変化を含んでいる場合を考える。

Example 3.2 不動産屋で新たに家が売れた。家と購入客を入力し、販売実績に加える

とする。この操作のスキーマ *(AddSold)* を考える。

$$
\begin{array}{|l}
\hline
AddSold \\
\Delta Estate \\
s? : Houses \\
c? : Customers \\
msg! : String \\
\hline
s? \notin dom\ sold \\
sold' = sold \cup \{s? \mapsto c?\} \\
msg! = \text{"}Thanks\ a\ lot.\text{"} \\
Houses' = Houses \\
\hline
\end{array}
$$

プライム (′) は，操作実行後の変数値を意味する。宣言部の$\Delta Estate$ は，Estate のすべての変数に対し、プライム無し，プライム付きの両方を宣言していることを表す。ここに、プライム無し変数はスキーマ評価前の値、プライム付き変数はスキーマ評価後の値を意味している。?付きの名称は入力変数を表し、!付きの名称は出力変数を表す。他の多くのデータ型, 表記法等、さらなるZの機能にはここでは触れず、文献 [33, 39, 8] に委ねることとする。

3.3 洗練化計算 (Refinement Calculus)

一般に知られている洗練化計算 [31, 39] は、仕様文 (specification statements) を使って洗練化操作を進める。仕様文は以下の形式から成る。

$$< frame >: [< precondition >, < postcondition >]$$

これは、$< precondition >$が満たされた場合にこの操作が実行され、$< postcondition >$ を満足する状態で終了することを意味している。ただし、$< frame >$にリストされている変数のみがこの操作で変化することを意味する。

Example 3.3

$$result! : [point \geq 50, result! = \text{"}pass\text{"}]$$

これは、変数 *point* が *50* 以上の時に、*result!*に*"pass"*を出力することを表している。

洗練化計算におけるプログラムとは、仕様文とコード（護衛付きコマンド文 [7] など）を混在させたものから成る。例えば、

$$x, y : \left[\begin{array}{ll} x = X & x' = X - Y \\ y = Y\ , & y' = X \end{array} \right] ; \ x := y + 1$$

は，仕様文が評価された後、代入文 $x := y + 1$ が実行されることを意味する。例えば $x = 3 \wedge y = 1$ の時は、仕様文の評価後 $x = 2 \wedge y = 3$ となり、その直後の代入文で、$x = 4$ に変化する。

これらプログラムに洗練化 (refinement) を導入する。プログラムPがプログラムQによって洗練化される (be refined) ことを、$P \sqsubseteq Q$ と表す。これは、以下の規則によって定義される。規則中のwは$< frame >$に対応する変数リストである。

Law 1　$post2 \Rightarrow post1$ ならば $w : [pre, post1] \sqsubseteq [pre, post2]$

すなわち、$post2$ が $post1$ より強い条件ならば、$[pre, post1]$ は $[pre, post2]$　に洗練化される。

Law 2　$pre1 \Rightarrow pre2$ ならば $w : [pre1, post] \sqsubseteq [pre2, post]$

すなわち、$pre1$ が $pre2$ より強い条件ならば、$[pre1, post]$ は $[pre2, post]$　に洗練化される。

Example 3.4 例えば、$x : [x > 10, x'^2 = 121] \sqsubseteq [x > -20, x'^2 = 121]$ が成り立つ。なぜなら、$pre1 \Rightarrow pre2$ を満たすからである。

Law 3 代入文の法則。$pre \Rightarrow post[E/w', x/x']$ なら、$w, x : [pre, post] \sqsubseteq w := E$

ただし、$post[E/w']$ は，変数 w' を E で置き換えた式を表す。上式で x は $post$ に現れる w 以外の変数全体である。

Example 3.5 例えば、$w : [x > 10, w > 11] \sqsubseteq w' := x + 1$ が成り立つ。なぜなら、$x > 10 \Rightarrow x + 1 > 11$ すなわち、$pre \Rightarrow post[E/w']$ を満たすからである。

Law 4 連接の法則。、

$$w, x : [pre, post] \sqsubseteq \left\{ \begin{array}{ll} begin & con X \bullet \\ & x : [pre, mid]; \\ & w, x : [mid[X/x, x/x'], post[X/x]] \\ end & \end{array} \right.$$

ただし、mid には x 以外の事前変数を含まないこと。

Example 3.6 例えば、この連接の法則より、

$$\begin{array}{rl} & w : [w = X, w = X^4] \\ \sqsubseteq & w : [w = X, w = X^2]; \\ & w : [w = X, w = X^2] \end{array}$$

が成り立つ。

Law 5 選択の法則。$pre \Rightarrow (G_1 \vee G_2 \vee \ldots \vee G_n)$ が成り立つとき、

$$w : [pre, post] \sqsubseteq \left\{ \begin{array}{ll} if & G_1 \rightarrow w : [G_1 \wedge pre, post] \\ \Diamond & G_2 \rightarrow w : [G_2 \wedge pre, post] \\ : & \\ \Diamond & G_n \rightarrow w : [G_n \wedge pre, post] \\ fi & \end{array} \right.$$

ただし、$if \ldots fi$ は護衛付きコマンドにおける選択構文である。

Law 6 反復の法則。

$$w : \left[\begin{array}{ll} & \neg G[w'/w] \\ inv, & inv[w'/w] \end{array} \right] \sqsubseteq \left\{ \begin{array}{ll} do & \\ & G \rightarrow w : \left[\begin{array}{ll} G & \neg G[w'/w] \\ inv, & o \leq V[w'/w] < V \end{array} \right] \\ od & \end{array} \right.$$

ただし、V は非負整数値をとる適当な関数とする。すなわち、$V[w'/w] < V$　は V の値が減少し、反復が必ず停止することを意味している。

Example 3.7

$$w, k : \left[\begin{array}{ll} & , \neg k' < 10 \\ 0 \leq k \leq 10 & , 0 \leq k \leq 10 \\ w = k! & , w' = k'! \end{array} \right]$$

ただし!は階乗を表す。この式は、上の *Law6* から以下の護衛コマンド文に洗練化される。

$$\sqsubseteq \left\{ \begin{array}{l} do \\ \qquad k < 10 \rightarrow w, k : \left[\begin{array}{ll} k < 10 & , \neg k' < 10 \\ 0 \leq k \leq 10 & , 0 \leq k \leq 10 \\ w = k! & , w' = k'! \\ & , k' = k + 1 \\ & , 0 \leq 10 - k' < 10 - k \end{array} \right] \\ od \end{array} \right.$$

反復は $0 \leq k \leq 10 \wedge \neg k < 10$ なる条件を満たして終了するので、終了時点では，$w = 10!$ を満足している。

Ｚは事前と事後の変数を区別して表現するので、洗練化計算として表現し易いという特徴を持つ。Ｚの表現の下で、本節で紹介した法則を用いて洗練化を進めることが可能であるが、一段階の洗練化では、一つのスキーマを対象として洗練化することになる。このことは洗練化を進める上での制限でもある。なぜなら、複数のスキーマを組み合わせて洗練化する方が考えやすい場合があるからである。

4　IO 正則式の形式的仕様への適用

4.1　IO 正則式

プログラムにはデータ間の関係を表現する部分と実行順序の制御を表現する部分の両面が含まれている。前者は，静的な（事前状態と事後状態）の論理関係で表現可能であるが、実行順序の制御に関しては、従来からの数学にはそれに匹敵する表現はほとんど存在しない。一般の手続き的プログラミング言語では，残念ながら，この両方を区別して表現することはしない。すなわち、複数行に記述された文の列は、連接であり、数式の羅列のように評価の順序が規定されていないものとは異なる。しかし、一般に分析段階では、操作の順序（すなわちアルゴリズム）は極力伏せられ，設計段階では、静的な部分と動的な（アルゴリズム的な）部分が分離されることが期待されている。そのような設計手法として，ジャクソン法 (JSP)[21] があり，ジャクソン法の表現ツールとしてジャクソン構造図がある。ジャクソン構造図は正規表現に変換することができ、最下層の箱は、データの操作（入出力，演算）を意味する。すなわち、、データ操作を表現する基本要素をもつ正規表現は，データ操作と実行順序の両方を区別して表現する記述法を与えていることになる。本論文で述べる IO 正則式 (Input Output Regular Expression) は、データ間の静的関係部分を基本要素として表現し、それらの動的な実行順序関係を正則式で表現するものである。ジャクソン（Jackson,M.) は，対象とする問題を，入力，出力のデータ構造の関係によって、構造一致 (structured corresponding) 問題と，不一致 (structured clash) 問題 [46] に分類した。構造一致問題は、入出力データ構造がそのままプログラム構造に反映されるケースであり，この場合には，入出力データ関係を IO 正則式で表現することによって、その表現がプログラムの構造にもなる。一方、構造不一致のケースは，入出力データ構造からプログラム構造が直接的に導かれないので，入出力データを基に IO 正則式で表現された仕様は、洗練化されなければ，プログラムの構造を表現することにはならない。

入出力データに基づいて状態を推移させるシステムに対しては、入出力ストリームの関係としてシステムの構造が表現される。しかし、個々の入力に対応して随時変化する

状態をストリームというデータ構造で表現しておくことは困難である。ストリームからストリームへの関数やストリーム間の関係の論理式を記述することがいかに困難であるかは [43, 2]、宣言的言語の利用を経験した人であれば容易に理解できる。筆者は、ＩＯ正則式を、データフローネットワークのプロセスにおけるストリーム関係を表現する記法として用いてきた。Ｚなどの仕様記述言語が普及してきた今、さらにその表現力と可能性が一般のシステムの仕様記述，洗練化へと広がってきたと考えられる。

IO 正則式 (I/O regular expression) は以下の定義で与えられる。

Definition 4.1

$$
\begin{aligned}
\langle factor \rangle &::= \langle primary\, term \rangle \mid [\,\langle term \rangle\,] \\
&\quad \mid \langle primary\, term \rangle^* \mid [\,\langle term \rangle\,]^* \\
\langle seq\ factor \rangle &::= \langle\ factor \rangle \mid \langle\ factor \rangle \langle seq\ factor \rangle \\
\langle term \rangle &::= \langle seq\ factor \rangle \mid \langle term \rangle \sqcup \langle seq\ factor \rangle \\
\langle IO\ regular\ expression \rangle &::= \langle term \rangle
\end{aligned}
$$

優先度は、高い方から、反復 $(*)$，連接 $(seq\, factor)$，選択 $(\sqcup)$ の順とする。同一順に対しては、右結合とする。$< primary\ term >$は、一般の述語論理式に事前事後の変数や入出力変数を含むことを許した表現であるとする。ただし以下の章では、$< primary\ term >$はＺで許されるスキーマを充てる。

直感的に，$\langle seq\ factor \rangle$ は，連接であり、順次評価することを意味する。$\langle term \rangle \sqcup \langle seq\ factor \rangle$ は選択であり、条件を満たすいずれかを評価することを意味する。$\langle primary\ term \rangle^*$ および $[\langle term \rangle]^*$は反復であり、論理式の評価結果が真である限り評価が繰り返される。ただし、これらの正確な意味は、後の章で、while プログラムによって与えることになる。$\sqcup$ についても、 基本的には非決定性であるが、後の意味付けでは、決定性に限定して扱うことになる。

以下では、IO 正則式の中心部だけを示すために、スキーマの代わりにそのスキーマの述語部のみを記述することがある。

Example 4.1 次の *IO* 正則式は入力チャネル *in* からの非負整数をすべて出力チャネル *out* に出力することを意味する。

$$[in? > 0 \wedge out! = in?]^*\ [in? \leq 0]$$

但し入力のストッパは 0 以下の整数値としている。?は入力変数を!は出力変数を表す装飾である。またプライム (′) 付き変数は、評価後の変数を意味する。これらは、後述のＺにおける記法と同様である。

Example 4.2 次の式は、出力チャネル *out* に数値列 *(1,2,...,100)* を生成するプログラムを表す。

$$[out! = k' \wedge k' = 1]\ [out! = k' \wedge k' = k + 1\ \wedge\ k < 100]^*$$

Example 4.3 次の式は、コンピュータが乱数によって数を決め、プレイヤーの入力した予想値と比較し、一致するまで繰り返すプログラムを意味する。

$$[numb' = rand()]$$

$$[[in? > numb \wedge msg! = "too\ large"] \sqcup [in? < numb\ \wedge\ msg! = "too\ small"]]^*$$

$$[in? = numb \wedge msg! = "congratulation"]$$

システムが入出力を伴うものであれば、仕様の段階から入出力に依存した流れのシステムの振る舞いを記述する必要がある。その場合に IO 正則式が有効である。なぜなら、IO 正則式では、個々の入力に伴うシステムの状態変化や、個々の状態における出力を記述することに適しているからである。一方、内部の状態変化が中心となるシステムにおいては、実行の順序を解析する前に、システム全体について、開始状態と終了状態の関係が必要となる。その場合は、設計に移行していく時にアルゴリズム（すなわち状態の動的変化）に順次変換していく必要があり、IO 正則式が利用できる。このように、IO 正則式は仕様や設計に対する表現として有効である。

IO 正則式を用いずに各基本項の実行順序を表現するためには、各基本項内に、その項が実行される為の条件を変数を用いて表現する必要がある。例えば、$S = [nxt = 1 \wedge n? >$

$0 \wedge out! = n? \wedge nxt' = 2]$ と $T = [nxt = 2 \wedge n? > 0 \wedge nxt' = 3]$ の場合、T は変数 $nxt = 2$ という事前条件を含むので、S の次に実行されることになる。$[S][T]$ なる連接として表しても勿論同じ振る舞いがなされるが、連接で表現する場合には、$next = 2$ などの事前条件を含めなくともよいことになる。この事は利点であるが、S や T が単独でそれらが実行されるべき条件を含んでいるので、記述されている位置に腐心することがない。すなわち、この項が実行されるのは、含まれている事前条件の場合のみであることが明示されている。一方このような事前条件を含まない記法では、その基本項の IO 正則式内での位置に依存して異なる条件下で実行されることに注意する必要がある。

4.2 従来からの洗練化との比較

Z 等の形式的仕様に基づく一般の洗練化 [31, 39] では、以下のような段階を経て操作が進行する。

1. 仕様（スキーマによる表現）⇒
2. 設計 (スキーマによる表現）⇒（抽象プログラム）⇒
3. コード (護衛付きコマンドプログラム）⇒
4. コード (プログラミング言語)

抽象プログラムは、仕様文 (specification statement) による表現である。仕様文はスキーマと護衛付きコマンドの混在した表現である。すなわちこのような洗練化では、4 通りの表記法が用いられることになる。

筆者の IO 正則式の下での洗練化手法は以下の過程を経て進められる。

1. 仕様 (IO 正則式)⇒
2. 設計 (IO 正則式)⇒
3. コード (プログラミング言語)

即ち仕様から設計までを一貫した表現によって行うことが出来る。スキーマの複合した部分を洗練化の対象とすることができるので、洗練化の候補を見いだしやすいという

利点がある。さらに、if 文や while 文の条件などは具体化せずに，最後のプログラミングの段階に残しているのが特徴である。ただし、洗練化計算では、仕様文を用いる洗練の規則があれば護衛付きコマンドプログラムまたはプログラミング言語によるプログラムへ到達できるのに対し、筆者の方法では、洗練化の段階には洗練化規則を用い、プログラムの正当性検証の段階はまた異なる規則を用いる必要がある。

4.3 Z 上の IO 正則式の性質

IO 正則式による洗練化を考えるために、まず Hoare 論理の記法に現れるプログラムの代わりに，Z のスキーマを基にしたＩＯ正則式を置いた関係を定義する。以下では、p, q は述語論理式（Z の述語部として許される表現で、プライム付き変数, 入力変数, 出力変数を含まないものを以下では述語論理式という），$[S]$ は IO 正則式における基本項（すなわち S はスキーマ名），$[\alpha], [\beta]$ などは、IO 正則式を表すものとする。

Definition 4.2 *IO* 正則式 $[\alpha]$ に対する事前条件 *(precondition)* を $pre([\alpha])$ で表し、以下で定義する。

- 基本項 $[S]$ の事前条件 *(precondition)*. スキーマ S の宣言部に含まれるプライム付き変数の全体を A', 出力変数の全体を $O!$とするとき、

$$pre([S]) = \exists A'; \exists O! \bullet S$$

(ただし S でスキーマ S の述語部を表している)

- 連接 $[\alpha][\beta]$ に対しては、

$$pre([\alpha][\beta]) = pre([\alpha])$$

- 選択 $[\alpha 1] \sqcup [\alpha 2]$ に対しては、

$$pre([\alpha 1] \sqcup [\alpha 2]) = pre([\alpha 1]) \cup pre([\alpha 2])$$

- 反復 $[\alpha]^*$ に対しては、

$$pre([\alpha]^*) = true$$

をそれぞれ意味するものとする。

Example 4.4 たとえば、$S \equiv x > 0 \wedge y > 1 \wedge x' = x + y$ であれば、$pre([S]) \equiv \exists x' \bullet x > 0 \wedge y > 1 \wedge x' = x + y$ すなわち $pre([S]) \equiv x > 0 \wedge y > 1$ である。

Definition 4.3 S をZスキーマ $[S]$ の述語部とする。

- $p \Rightarrow pre([S])$ かつ $(p \wedge \hat{S}) \Rightarrow q'$ のとき、$\{p\}[S]\{q\}$ と表す。
- $p \Rightarrow pre([S])$ かつ $(\exists A \bullet p \wedge \hat{S}) \equiv q'$ のとき、q は条件 p の下での $[S]$ の最強事後条件 *(strongest post condition)* とよび、$q = sp(\{p\}, [S])$ と記す。

ただし、q'は、述語論理式 q に対し、$\hat{S}$内にプライム付きとして宣言されている変数のみをすべてプライム付きに変換した式を表す。$\hat{S}$は、スキーマ $[S]$ に入力変数 $x?$や出力変数 $w!$が宣言されているとき、列 X, W に対する式 $X' = X^\frown x?$や $W' = W^\frown w!$を S に加えた述語とする。A は S 内にプライム付き，プライム無しの両方で宣言されている変数の集合とする。

一般のIO正則式に対する最強事後条件については、後述する。

たとえば、スキーマ $[S]$ の述語部を $a' = a + b$ とすれば、$\{a = b \wedge a' = a + b\} \Rightarrow \{a' = 2 * b\}$ が成り立つので、$\{a = b\}[S]\{a = 2 * b\}$ である。また、$\{\exists a : a = b \wedge a' = a + b\} \equiv \{a' = 2 * b\}$ が成り立つので、$sp(\{a = b\}, [S]) = \{a = 2 * b\}$ である。

Definition 4.4 *IO* 正則式 $[\alpha]$ に対し、適当な非負整数関数 $V(u)$ が存在して、

$$\{pre([\alpha]) \wedge V(\vec{u}) = k\}[\alpha]\{V(\vec{u}) < k\}$$

が定義されるとき、$dcond([\alpha])$ と表す。ただし$\vec{u}$は $[\alpha]$ に現れる変数のベクトル，k は $[\alpha]$ に現れない変数とする。

この条件 *dcond* は，IO 正則式が反復されるときに有限回の反復で停止するための条件である。IO 正則式の評価のあとに、非負関数 $V(u)$ の値が減少するのであれば、反復した場合に必ず停止することを意味する。

Law 1 以下で $[\alpha 1], [\alpha 2]$ などは *IO* 正則式，p, q, r, w は *(プライム付き変数を含まない)* 述語論理式とする。

1. 連接の規則：

$$\frac{\{p\}[\alpha 1]\{q\} \ \ \{q\}[\alpha 2]\{r\}}{\{p\}[\alpha 1][\alpha 2]\{r\}}$$

2. 選択の規則：$pre([\alpha 1]) \wedge pre([\alpha 2]) = false, p \Rightarrow pre([\alpha 1]) \vee pre([\alpha 2])$ を満たし，$pre([\alpha 1]) = w$ とするとき，

$$\frac{\{p \wedge w\}[\alpha 1]\{r\} \ \ \{p \wedge \neg w\}[\alpha 2]\{r\}}{\{p\}[\alpha 1] \sqcup [\alpha 2]\{r\}}$$

3. 反復の規則：$pre([\alpha]) = w$ とするとき，

$$\frac{\{p \wedge w\}[\alpha]\{p\} \ dcond([\alpha])}{\{p\}[\alpha]^*\{p \wedge \neg w\}}$$

4. 帰結の規則:

$$\frac{p \Rightarrow p1 \ \ \{p1\}[\alpha]\{q1\} \quad q1 \Rightarrow q}{\{p\}[\alpha]\{q\}}$$

従って、これらの規則は，ループの停止性を保証していることになるので、完全正当性 [45] に相当するものである。

Example 4.5 $[S1]$ の述語部を $a' = a + b$ とすれば $\{a = b\}[S1]\{a = 2 * b\}$ が成り立ち、$[S2]$ の述語部を $a' = a + 1$ とすれば $\{a = 2 * b\}[S2]\{a = 2 * b + 1\}$ が成り立つ。
故に連接の規則より、$\{a = b\}[S1][S2]\{a = 2 * b + 1\}$

Example 4.6 $S1 \equiv a > 0 \wedge a' = a + b$，$S2 \equiv a \leq 0 \wedge a' = a + 1$ とする。このとき、$pre([S1]) \cap pre([S2]) = false$ である。$\{a = b \wedge a > 0\}[S1]\{a > b\}$ $\{a = b \wedge a \leq 0\}[S2]\{a > b\}$ より、選択の規則によって $\{a = b\}[S1] \sqcup [S2]\{a > b\}$ が成り立つ。

Example 4.7 $S \equiv a \leq 10 \wedge a' = a + 1$ のとき、$pre(S) \equiv a \leq 10$ である。非負の関数 $V = 10 - a$ に対し、$\{a \leq 10 \wedge 10 - a = k\}[S]\{10 - a < k\}$ が成り立つので、$dcond([S])$ を満たす。さらに、$\{a \leq 11 \wedge a \leq 10\}[S]\{a \leq 11\}$ も成り立つ。故に反復の規則より、$\{a \leq 11\}[S]^*\{a \leq 11 \wedge a > 10\}$ すなわち、$[S]$ の反復後 a は *11* の値をとる。

Definition 4.5 任意の p,q に対し，$\{p\}[\alpha]\{q\} \Leftrightarrow \{p\}[\beta]\{q\}$ のとき、そのときに限り $[\alpha] \equiv [\beta]$　と表す。

上の規則，定義より、IO 正則式の基本的な性質が導かれる。

Lemma 4.6 $[\alpha 1], [\alpha 2], [\alpha 3], [\beta]$　を任意の *IO* 正則式とするとき以下が成り立つ。

1. $[\alpha 1] \sqcup [\alpha 2] \equiv [\alpha 2] \sqcup [\alpha 1]$

2. $([\alpha 1] \sqcup [\alpha 2]) \sqcup [\alpha 3] \equiv [\alpha 1] \sqcup ([\alpha 2] \sqcup [\alpha 3])$

3. $([\alpha 1][\alpha 2])[\alpha 3] \equiv [\alpha 1]([\alpha 2][\alpha 3])$

4. $([\alpha 1] \sqcup [\alpha 2])[\beta] \equiv [\alpha 1][\beta] \sqcup [\alpha 2][\beta]$

proof: 1 の証明:$\{p\}[\alpha 1] \sqcup [\alpha 2]\{r\}$ が成り立つとすれば、選択の規則より，$w = pre([\alpha 1])$ にして，$\{p \wedge w\}[\alpha 1]\{r\}$ と $\{p \wedge \neg w\}[\alpha 2]\{r\}$ が成り立っている。$p \wedge \neg w \equiv p \wedge pre([\alpha 2])$ が成り立つので、同じ関係が得られ，$\{p\}[\alpha 2] \sqcup [\alpha 1]\{r\}$　も成り立つ。

2 の証明:左辺⇒ 右辺を示す。左辺が成り立つとすれば、選択の規則より，$w = pre([\alpha 2] \sqcup [\alpha 1])$ に対して，

$$\{p \wedge w\}[\alpha 1] \sqcup [\alpha 2]\{r\} \tag{1}$$

$$\{p \wedge \neg w\}[\alpha 3]\{r\} \tag{2}$$

が成り立っている。式 *1* は選択の規則より、$v = pre([\alpha 1])$ に対して，

$$\{p \wedge w \wedge v\}[\alpha 1]\{r\} \tag{3}$$

$$\{p \wedge w \wedge \neg v\}[\alpha 2]\{r\} \tag{4}$$

が成り立つ。上式 *2* の事前条件を強めても同じ事後条件を満たすので、以下が成り立つ。

$$\{p \wedge (\neg w \cup \neg v) \wedge \neg w\}[\alpha 3]\{r\} \tag{5}$$

$\{p \wedge w \wedge \neg v\} \equiv \{p \wedge (\neg w \wedge \neg v) \wedge w\}$　が成り立つので、式 *4* は以下の式と同じである。

$$\{p \wedge (\neg w \wedge \neg v) \wedge w\}[\alpha 2]\{r\} \tag{6}$$

式 *5* と *6* に選択の規則を適用して、以下が成り立つ。

$$\{p \wedge (\neg w \wedge \neg v)\}[\alpha 2] \sqcup [\alpha 3]\{r\} \tag{7}$$

従って、式 *3* と *7* に選択の規則を適用して、以下が成り立つ。

$$\{p\}[\alpha 1] \sqcup ([\alpha 2] \sqcup [\alpha 3])\{q\} \tag{8}$$

左辺 ⇐右辺も同様に示される。**??**の証明：省略。

4 の証明:左辺⇒ 右辺を示す。左辺に対して、以下が成り立つとする。

$$\{p\}[\alpha 1][\beta] \sqcup [\alpha 2][\beta]\{q\} \tag{9}$$

すなわち、$w = pre([\alpha 1])$ に対し、

$$\{p \wedge w\}[\alpha 1][\beta]\{r\} \tag{10}$$

$$\{p \wedge \neg w\}[\alpha 2][\beta]\{r\} \tag{11}$$

が成り立つ。連接の規則から、上式 *10,11* は，それぞれ、適当な $q1, q2$ が存在して、以下の式になる。

$$\{p \wedge w\}[\alpha 1]\{q1\} \qquad \{q1\}[\beta]\{r\} \tag{12}$$

$$\{p \wedge \neg w\}[\alpha 2]\{q2\} \qquad \{q2\}[\beta]\{r\} \tag{13}$$

帰結の規則より、以下の２式が得られる。

$$\{p\}[\alpha 1][\alpha 2]\{q1 \vee q2\} \qquad \{q1 \vee q2\}[\beta]\{r\} \tag{14}$$

故に

$$\{p\}([\alpha 1] \sqcup [\alpha 2])[\beta]\{r\} \tag{15}$$

が成り立ち、左辺⇒ 右辺が示された。つぎに，左辺 ⇐右辺を示すために上の関係 *15* を仮定する。このとき、適当な q が存在し、

$$\{p\}[\alpha 1] \sqcup [\alpha 2]\{q\} \qquad \{q\}[\beta]\{r\} \tag{16}$$

選択の規則より、

$$\{p \wedge w\}[\alpha 1]\{q\} \tag{17}$$

$$\{p \wedge \neg w\}[\alpha 2]\{q\} \tag{18}$$

が成り立つので，

$$\{p \wedge w\}[\alpha 1][\beta]\{r\} \tag{19}$$

$$\{p \wedge \neg w\}[\alpha 2][\beta]\{r\} \tag{20}$$

も成り立つ。よって、左辺 ⇐右辺が示された。

なお、$[\beta]([\alpha 1] \sqcup [\alpha 2]) \equiv [\beta][\alpha 1] \sqcup [\beta][\alpha 2]$ は成り立たない。なぜなら、$pre([\beta][\alpha 1])$ と $pre([\beta][\alpha 2])$ が背反でないので右辺が定義されない為である。

Lemma 4.7 $[S]$ を任意のスキーマ，p, q を任意の述語論理式とするとき、以下が成り立つ。

$$\{p\}[S]\{q\} \quad \equiv \quad sp(p, [S]) \Rightarrow q$$

これは、スキーマに対する最強事後条件である。さらに IO 正則式の各構造に対する最強事後条件を以下で定義する。

Definition 4.8 任意の *IO* 正則式を $[\alpha]$ とする。以下の定義を満たす述語論理式 $r = sp(p, [\alpha])$ を、p の下での $[\alpha]$ の最強事後条件 *(strongest postcondition)* という。

- 連接に対する定義：

  $sp(p, [\alpha][\beta]) = sp(sp(p, [\alpha]), [\beta])$

- 選択に対する定義：

  $sp(p, [\alpha] \sqcup [\beta]) = sp(p \wedge pre([\alpha]), [\alpha]) \vee sp(p \wedge pre([\beta]), [\beta])$

- 反復に対する定義：

  $sp(p, [\alpha]^*) = \neg pre([\alpha]) \wedge \bigcup_{0 \leq i} q_i$

  ただし、q_iは以下で定義される $q_i (i = 0, 1, 2 \ldots)$。

  $q_0 = p$ とし、
  $\neg(q_i \Rightarrow \neg pre([\alpha]))$ のとき $q_{i+1} = sp(q_i \wedge pre([\alpha]), [\alpha])$
  $q_i \Rightarrow \neg pre([\alpha])$ のとき $q_{i+1} = q_i$

この定義とスキーマに対する最強事後条件の定義より、次の性質が成り立つ。

Lemma 4.9 $[\alpha]$ を $dcond([\alpha])$ を満たす任意の *IO* 正則式，p, q を任意の述語論理式とするとき以下が成り立つ。

$$\{p\}[\alpha]\{q\} \equiv sp(p, [\alpha]) \Rightarrow q$$

*proof:*構造に関する帰納法で証明する。
連接に対する証明： $[\alpha], [\beta]$ に対しこれが成り立っていると仮定する。このとき
$\{p\}[\alpha][\beta]\{r\} \Longleftrightarrow \quad \exists q : \{p\}[\alpha]\{q\}\ \{q\}[\beta]\{r\} \Longleftrightarrow$
$\exists q : sp(p, [\alpha]) \Rightarrow q \quad sp(q, [\beta]) \Rightarrow r \Longleftrightarrow$
$sp(sp(p, [\alpha]), [\beta]) \Rightarrow r \Longleftrightarrow sp(p, [\alpha][\beta]) \Rightarrow r$
故に連接に対し成り立つ。

選択に対する証明：$\{p\}[\alpha] \sqcup [\beta]\{r\}$ を仮定する。これが成り立つためには、選択の規則の前提が満たされていなければならない。すなわち、$\{p \wedge pre([\alpha])\}[\alpha]\{q\}$ かつ

$\{p \wedge \neg pre([\alpha])\}[\beta]\{q\}$ が成り立つ。帰納法の仮定より、これらは最強事後条件を使って、$sp(p \wedge pre([\alpha]),[\alpha]) \Rightarrow q$ および、$sp(p \wedge \neg pre([\alpha]),[\beta]) \Rightarrow q$ と表される。後者は、選択の規則の前提条件より、$pre([\alpha]) \wedge pre([\beta]) = false, p \Rightarrow pre([\alpha]) \vee pre([\beta])$ が成り立つので、$sp(p \wedge pre([\beta]),[\beta]) \Rightarrow q$ に等価である。こうして、$sp(p,[\alpha] \sqcup [\beta]) \Rightarrow q$ が成り立つことになる。逆についても同様に証明される。

反復に対する証明：$\{p\}[\alpha]^*\{q\}$ を仮定すると、反復の規則より、適当な述語 $p1$ が存在し、

$$p \Rightarrow p1 \quad \{p1\}[\alpha]^* \quad \{p1 \wedge \neg pre([\alpha])\} \quad p1 \Rightarrow q$$

真ん中の性質は，反復の規則の前提から、$\{p1 \wedge pre([\alpha])\}[\alpha]\{p1\}$ を満たしていなければならない。これは、帰納法の仮定より、$sp(p1 \wedge pre([\alpha]),[\alpha]) \Rightarrow p1$ を満たす。一方

$sp(p,[\alpha]^*) = \neg pre([\alpha]) \wedge \bigcup_{0 \leq i} q_i$

であり、各 q_{i+1}は、$q_{i+1} = sp(q_i \wedge pre([\alpha]),[\alpha])$ である。故に直前に導いた性質より、$q_{i+1} \Rightarrow p1$ が成り立つ。故に、$sp(p,[\alpha]^*) \Rightarrow \{p1 \wedge \neg pre([\alpha])\}$ が示された。従って $sp(p,[\alpha]^*) \Rightarrow q$ が成り立つ。逆も同様に示される。

4.4 IO 正則式の下での洗練化

洗練化は、IO 正則式を順次プログラムに近づけていく操作である。以下の規則によって、IO 正則式の各項を具体化していくことができる。ただし、以下では、ＩＯ正則式が $dcond()$ 条件を満たしているとが証明済みとして扱う。

Definition 4.10 *(洗練化の定義)*

$[\alpha],[\beta]$ を任意の *IO* 正則式とする。

$pre([\alpha]) \equiv pre([\beta])$ かつ $sp(pre([\alpha]),[\beta]) \Rightarrow sp(pre([\alpha]),[\alpha])$ が成り立つとき、$[\alpha]$ は $[\beta]$ によって洗練化される *(refined)* といい、

$$[\alpha] \sqsubseteq [\beta]$$

と表す。

前節の洗練化計算においては、仕様文が使われ、その中には、frame 部分すなわち、影響する変数のリストが含まれていた。すなわち、この frame 部分に現れない変数に関しては、変化しないことを意味している。これは、Ｚスキーマにおける宣言部に, プライム付き変数と出力変数の形で含まれている。従って、我々のＺスキーマを基本項とする洗練化では、frame に相当する部分は敢えて表面に記述することはしない。

この定義 4.10 を基に、項を洗練化するために有効な，いくつかの性質を揚げる。

Example 4.8 $S \equiv a > 1 \wedge a' = 2 * a, S1 \equiv a > 1 \wedge a' = 3 * a$ とする。$a > 3 \equiv sp(a > 1,[S])$ かつ $a > 2 \equiv sp(a > 1,[S1])$ であり、$a > 3 \Rightarrow a > 2$ であるから、

$$[S] \sqsubseteq [S1]$$

明らかに、洗練化は推移律 (transitive law) を満たす。即ち、以下が成り立つ。

Lemma 4.11 *(推移律)*

$[\alpha 1],[\alpha 2],[\alpha 2]$ を任意の *IO* 正則式とするとき、$[\alpha 1] \sqsubseteq [\alpha 2] \wedge [\alpha 2] \sqsubseteq [\alpha 3]$ ならば $[\alpha 1] \sqsubseteq [\alpha 3]$ が成り立つ。

Example 4.9 $S \equiv a > 1 \wedge a' = 6 * a, S1 \equiv a > 1 \wedge a' = 2 * a$, $S2 \equiv a' = 3 * a$ とする。$a > 6 \equiv sp(a > 1,[S])$ かつ $a > 6 \equiv sp(a > 1,[S1][S2])$ であるから、

$$[S] \sqsubseteq [S1][S2]$$

Lemma 4.12 （選択への洗練化）

述語論理式 G_1, G_2は、$G_1 \wedge G_2 \equiv false$ と $G_1 \vee G_2 \equiv true$ を満たし、含まれる変数はすべてスキーマ $[S]$ 内で宣言されているとする。このとき、

$$[S] \sqsubseteq [S \wedge G_1] \sqcup [S \wedge G_2]$$

が成り立つ。ここに、$[S \wedge G_1]$ は、$[S]$ の述語部に G_1を連言として加えた基本項を表す。

proof: $pre([S]) \equiv pre([S \wedge G_1]) \vee pre([S \wedge G_2]) \equiv pre([S \wedge G_1] \sqcup [S \wedge G_2])$ が成り立つ。これを p とおけば、 $sp(p, [S \wedge G_1] \sqcup [S \wedge G_2]) \Rightarrow sp(p, [S])$ が成り立つ。

Example 4.10 $S \equiv [b' = a]$ のとき、$G_1 \equiv a > 10$, $G_2 \equiv 10 \geq a$ とすれば、

$$[S] \sqsubseteq [a > 10 \wedge b' = a] \sqcup [10 \geq a \wedge b' = a]$$

Lemma 4.13 （反復への洗練化）入力出力変数を含まない基本項 $[S]$ が適当な述語論理式 u によって $S \equiv T \wedge \neg u$ と表され，$dcond(T)$ を満たすとする。このとき、$\{pre([T]) \wedge u\}[T]\{pre([T])\}$ かつ、$\{pre([T]) \wedge \neg u\}[T]\{\neg pre([T])\}$ を満たすなら、

$$[S] \sqsubseteq [\alpha][T]^*$$

が成り立つ。ただし $[\alpha]$ は $pre([S]) \equiv pre([\alpha])$ かつ $pre([S])[\alpha]pre([T])$ を満たす IO 正則式とする。

proof: $sp(pre([T]) \wedge \neg u, [S]) = sp(pre([T]) \wedge \neg u, [T]) = r$ となる述語論理式 r が存在する。反復の最後の実行は $\{pre([T]) \wedge \neg u\}[T]\{r\}$ である。すなわち、$\{pre([T])\}[T]^*\{r\}$ が成り立ち、$sp(pre([T]) \wedge \neg u, [T]^*) \Rightarrow sp(pre([T]) \wedge \neg u, [S])$ である。$\{pre([S])\}[\alpha]\{pre([T])\}$ であるから、$\{pre([S])\}[\alpha][T]^*\{r\}$. 故に $[S] \sqsubseteq [\alpha][T]^*$ が成り立つ。

Example 4.11 $S \equiv a \leq 6 \wedge a' = a + 1 \wedge \neg(a \neq 6)$ とする。このとき、$\{\neg a \neq 6\} = wp([S], \neg a \neq 7)$ が成り立つ。この S は単に $a' = 7$ を表しているに過ぎない。$T \equiv a \leq 6 \wedge a' = a+1$ とおけば、$pre([T]) = a \leq 6$ であり、上の $lemma$ における u は、$u \equiv a \neq 6$ である。$\{pre([T]) \wedge u\}[T]\{pre([T])$ と $\{pre([T]) \wedge \neg u\}[T]\{\neg pre([T])$ が共に成り立ち、$dcond([T])$ は満たされる。さらに $\{a = 6\}[a = 6 \wedge a' = 3]\{pre([T])\}$ が成り立つので、以下がいえる。

$$[S] \sqsubseteq [a = 6 \wedge a' = 3][T]^*$$

もちろん、$a' = 3$ の代わりに $a' = -5, a' = 0, a' = 1$ など無数の IO 正則式を置くことができる。

本節の定義や $lemma$ を基にしてさらに洗練化の法則を導き出すことが可能である。

4.5 洗練化の例

ダイクストラの文献 [7] において、洗練化の例として用いられている問題を筆者の方法で洗練化する。文献 [7] においては、非形式的な方法で洗練化を進めているものである。

Example 4.12 整数配列 $s[1..N]$ に対し、$0 \leq i \leq j \leq N$ の部分和 $\sum_{k=i}^{k=j} s_k$ のうち、最小の値を求める。

まずスキーマ $MSdata$ は，計算に必要な変数の宣言と基本的条件を表す。なおZでは配列を seq で表現している。

MSdata

$$s : seq\mathbb{Z}$$
$$x, y, n, N : \mathbb{N}$$

$$\#s = N$$
$$N = 100$$

共通関数の定義。$srsum\ k$ は、k 番め以下のすべての部分和の集合。$sm\ i\ j$ は、$i \leq j$ 間のすべての値の和。

[X]

$$s : seq\mathbb{Z}$$
$$srsum : X \rightarrow \mathbb{P}X$$
$$sm : X \times X \rightarrow X$$

$$\forall k : X \bullet srsum\ k = \{w : X \mid 0 \leq i \leq j \leq k \wedge w = (sm\ i\ j)\}$$
$$\forall i, j : X \mid 0 \leq i \leq j \bullet \quad sm\ i\ j =$$
$$((0 < j < n \Rightarrow (sm\ i\ j) = ((sm\ i\ (j-1)) + (s\ (j-1)))) \wedge$$
$$(j = 0 \Rightarrow (sm\ i\ j) = 0))$$

最小部分列を x に求める仕様の最初の表現 $MinSum$。$MinSum$ をその表現のまま計

算をすれば、時間計算量は $O(n^3)$ となる。

MinSum

$\Delta MSdata$

$x' = min(srsum\ N)$

MinSum を以下のスキーマ *ItMin* の反復形に洗練化する。*Initn* は初期化のためのスキーマ。

Initn

$\Delta MSdata$

$n' = N$

ItMin

$\Delta MSdata$

$0 \leq n \wedge n \leq N$
$x' = min(srsum\ n)$
$n' = n + 1$

Initz

$\Delta MSdata$

$n' = 0$

さらに、*ItRev* の反復に洗練化する。*Initall* は、初期設定スキーマ。

Initall

$\Delta MSdata$

$x' = 0$
$y' = 0$
$n' = 0$

ItRev

$\Delta MSdata$

$0 \leq n \wedge n \leq N$
$x' = min\{x, y\}$
$y' = min\{(s\ n) + y, 0\}$
$n' = n + 1$

これらスキーマによる *IO* 正則式の下での洗練化は以下の過程を経る。

$$
\begin{array}{lcll}
[MinSum] & \sqsubseteq & [Initn][ItMin] & \ldots\ldots(a) \\
[Initn]\,[ItMin] & \sqsubseteq & [Initn]\,[ItMin \wedge \{n = N\}] & \ldots\ldots(b) \\
[Initn]\,[ItMin \wedge \{n = N\}] & \sqsubseteq & [Initn][Initz][ItMin]^* & \ldots\ldots(c) \\
[Initn]\,[Initz][ItMin]^* & \sqsubseteq & [Initz][ItMin]^* & \ldots\ldots(d) \\
[Initz]\,[ItMin]^* & \sqsubseteq & [Initall][ItRev]^* & \ldots\ldots(e)
\end{array}
$$

ItMin の *n=N* の場合が *MinSum* になる。従って、定義 *4.10* より、$[Initn][ItMin]$ なるスキーマに洗練化され、*(a)* が成り立つ。さらに *lemma4.13* より、$[ItMin \wedge \{n = N\}] \sqsubseteq [ItMin]^*$ が成り立つので、*(c)* となる。先頭の $[Initn]$ を削除しても事前条件は変わらず、全く同じ意味のＩＯ正則式になるので、*(d)* が成り立つ。さらに、$pre([Initz]) = pre([Initall])$ と *lemma 4.9* より、*(e)* が成り立つ。

なお、洗練化の最終結果 $[Initall][ItRev]^*$ における各スキーマの述語部をもとにした *Pascal* 風プログラムへの変換によって、以下が得られる。変換したプログラムの *IO* 正則式に対する一般的な証明については後の節で述べられる。

```
x:=0; y:=0 ; n:= 0;
while  0 ≦ n ∧ n ≦ N do
   wx:=x; wy :=y;
   x:=min(wx,wy);
   y:=min(s[n]+wy,0);
   n:=n+1
endwhile
```

ちなみに，この $[Initall][ItRev]^*$における計算の時間計算量は、$O(n)$ であり、仕様を直接計算する場合に比べて格段の効率化が計られている。

なお、この問題を洗練化計算 *(refinement calculus)* によって洗練化を進めると次のようになる。

$$
\begin{array}{ll}
 & [MinSum] \\
\sqsubseteq & n := N;\ [ItMin \wedge \{n = N\}] \\
\sqsubseteq & n := N;\ n := 0;\ do\ 0 \leq n \wedge n \leq N\ \rightarrow\ [ItMin]\ od
\end{array}
$$

この段階までくると、IO 正則式による洗練化と同様のステップでの変換はできなくなる。なぜなら、洗練化計算では、逐次プログラムの文に変換をしてしまうので、さらにその文を利用して洗練化することはあり得ない，すなわち、常に洗練化の対象となるのは、1つのスキーマだからである。IO 正則式の洗練化では、プログラム文への変換は最終段階に残すので、このようなことはなく、この例の *(c)* に見られるように，各段階でスキーマの複合された式を正規表現の性質を用いて一括して洗練化することが可能である。

Example 4.13 ボイヤームーアのストリングマッチングアルゴリズムを洗練化によって導出する。

例えば、$p =$ "$learn$" , $t =$ "$those\_learning\_to\_program$" とし、文字列 p が t に存在するか否かを判定する。p の右端から左方向に比較をするので、まず、以下の位置から比較が進められる。

$$
\begin{array}{l}
learn \\
those\_learning\_to\_program \\
\quad\uparrow
\end{array}
$$

最初の比較で一致しないので、↑ の位置を右に移動する。このとき、t に置ける文字が p には存在する e であるので、共に文字 e になる位置まですなわち、3文字移動することができる。

$$
\begin{array}{l}
\quad learn \\
those\_learning\_to\_program \\
\quad\quad\uparrow
\end{array}
$$

この場合も e と n で一致しないので、やはり3文字進める。すると、一致する位置に達し、判定が *true* で終了する。一致しないとき、判定位置を移動させる文字数は、別途関数としてあらかじめ求めておく必要がある。なお、p 内に存在しない文字に対しての移動文字数は、p の文字数と同じである。

ストリングマッチングの仕様は以下のように表される。まず共通データの宣言である。

$$
\begin{array}{|l}
p, t : seq\mathbb{Z} \\
m, n : \mathbb{N} \\
\hline
n = \#t \\
m = \#p
\end{array}
$$

$[X]$

$$
\begin{array}{|l}
nonMT : (X \times X) \nrightarrow Bool \\
allNON : (X \times X) \nrightarrow Bool \\
\hline
\forall l, m : X \bullet nonMT\ l\ m = \exists k : X \mid l - m < k \leq l \bullet t[k] \neq p[k - l + m + 1] \\
\forall m, n : X \bullet allNON\ m\ n = \forall i : X \mid 1 \leq i \leq n - m \bullet (nonMT\ (m + i)\ m)
\end{array}
$$

$nonMT\ l\ m$ は、ストリング t の位置 l にストリング p の最後部を置いたとき、t 対応する部分が一致しないことを意味する。$allNON\ m\ k$ は、ストリング t の位置 k までに対し、ストリング p と一致する部分がないことを意味する。

求める結果は変数 ans に *true* または *false* で得られる。

$MchT$

$$
\begin{array}{|l}
ans' : Bool \\
\hline
\neg(allNON\ m\ n) \\
ans' = true
\end{array}
$$

$MchF$

$ans' : Bool$

$allNON\ m\ n$

$ans' = false$

IO正則式で表現した仕様は、以下になる。

$$[MchT] \sqcup [MchF]$$

スキーマ$MchT$を洗練化する。$MchT$の変数nをlとして，以下のスキーマ$AnMchT$が$l \leq n$において満たされないときの結果が$MchT$であるので、$AnMchT$の反復形に洗練化する。

$AnMchT$

$l, l' : \mathbb{N}$

$m \leq l \leq n$

$\forall i : \mathbb{N} \mid m < i \leq l \bullet nonMT\ i\ m$

$l' = l + 1$

$ATrue$

$l, l' : \mathbb{N}$

$ans' : Bool$

$l \leq n$

$ans' = true$

$MInit$

$l, l' : \mathbb{N}$

$l' = m$

これらを使って

$$[Minit][MchT] \sqsubseteq [Minit][AnMchT]^*[ATrue]$$

となる。$AnMchT$の反復は以下の$RepMchT$の反復と同じであるから、

$$[Minit][MchT] \sqsubseteq [Minit][RepMchT]^*[ATrue]$$

となる。

$RepMchT$

$l, l' : \mathbb{N}$

$m \leq l \leq n$

$\exists k : l - m < k \leq l \bullet t[k] \neq p[k - l + m + 1]$

$l' = l + 1$

比較位置の移動数を計算する関数mdを用いて、移動数を1から$md()$に変えると$RepMchT$は以下のスキーマ$MvMchT$になる。

故に、

$$[MchT] \sqsubseteq [Minit][MvMchT]^*[ATrue]$$

となる。

$MvMchT$

$l, l', r, r' : \mathbb{N}$

$m \leq l \leq n$

$nonMT\ l\ m$

$r' = md(t[k])$

$l' = l + r'$

ここに関数mdの定義は以下の通りである。

$[X]$

$md : X \to \mathbb{N}$

$md\ c = m \land c \notin p \lor md\ c = m - i \land c = p[i]$

一方 p と一致する部分が t 内に存在する場合も同様にスキーマを作ることができるが、両方の統合したものを作る。すなわち、同じスキーマの反復で、終了時の状態によって、$true$ か $false$ かを判定するようにスキーマを作る。

$ATrue$ の反対の結果に対するスキーマを $AFalse$ とすれば、

$AFalse$

$\Delta SMdec$
$l, l', r, r' : \mathbb{N}$
$ans : Bool$

$l > n$
$ans' = false$

$MchT$ と同様に $MchF$ に対しても、

$$[Minit][MchF] \sqsubseteq [Minit][MvMchT]^*[AFalse]$$

となる

最初の仕様 $[MchT] \sqcup [MchF]$ は、

$$[MchT] \sqcup [MchF] \sqsubseteq [Minit][[MchT] \sqcup [MchF]]$$

$$[MchT] \sqcup [MchF] \sqsubseteq [Minit][MvMchT]^* \, [\,[ATrue] \sqcup [AFalse]\,]$$

となる。

Example 4.14 有向グラフの各辺に重みがついているネットワークを考え、与えられた点 *(点 1 としている)* からその他の各点への路で重み和が最小になる値とそのときの路を求める。これはダイクストラのアルゴリズムが知られている。仕様からダイクストラのアルゴリズムへ洗練化することを試みる。グラフを $G(V, E)$，各辺の重みを表す関数 w とする。これらの保つべき性質は，以下のスキーマ $MPcond$ で与えられる。ここでは、

辺 E に存在しない (i, j) に対しては、あらかじめいかなる重みよりも大きい値 9999 を w の値として与えてある。

$MPcond$

$w : (\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$
$V : \mathbb{P}\mathbb{N}$
$E : \mathbb{P}(\mathbb{N} \times \mathbb{N})$
$bnd : \mathbb{N}$

$domw = V \times V$
$domE \subseteq V$
$ranE \subseteq V$
$(i, j) \in (dom\ w - E) \Rightarrow w(i, j) = 9999$
$(i, j) \in E \Rightarrow w(i, j) \leq bnd$

次は共通に利用される関係，関数の定義を与えている。辺の連なりすなわちパスを seq で表現している。$pthrel$ はグラフGに存在するパスであることを意味する関係。$cost$ はパスに対する重み値の和を求める関数。 $ncycl$ は $pthrel$ を満たしかつ、サイクルを含まないパスを意味する関係。 $Apth\ i\ j$ は i, j 間を結ぶ $ncycl$ を満たすパスの集合を求める関数。

$[VV, EE, Ww]$

$pthrel : \mathbb{P}(EE \times Ww \times seqEE)$
$cost : Ww \times seqEE \nrightarrow \mathbb{N}$
$Apth : EE \times Ww \times \mathbb{N} \times \mathbb{N} \nrightarrow \mathbb{P} seqEE$
$ncycl : \mathbb{P}(EE \times Ww \times seqEE)$

$(E, w, path) \in pthrel \Leftrightarrow$
$(\#path = 1 \land \exists (a, b) : E \bullet path\ 1 = (a, b))$
$\lor$
$(\#path > 1 \land \forall i : 1 \leq i < \#path;\ \exists (a, b), (b, c) : E$
$\bullet path\ i = (a, b) \land path\ (i + 1) = (b, c))$
$(E, w, path) \in pthrel \Rightarrow \quad cost\ w\ path = \sum_{i=1}^{i=\#path-1} w(path\ i)$
$(E, w, path) \in ncycl \Leftrightarrow$
$(E, w, path) \in pthrel \land$
$(path\ 1 = (a, b) \land path\ \#path = (c, d) \Rightarrow a \neq d) \land$
$\forall i, j : 1 \leq i, j \leq \#path$
$\mid i \neq j \land path\ i = (a, b) \land path\ j = (c, d)$
$\bullet a \neq c$
$Apth(E, w, i, j) = \{path \in seqEE \mid (E, w, path) \in ncycle \land$
$\exists q, r \in VV \bullet path\ 1 = (i, q) \land path\ \#path = (r, j)\}$

目的は関数 $D\ t$ の値がが点 1 から任意の点 t までのパスのコストの最小値になるようにすることである。それは以下の仕様 $MinPath$ で表される。

MinPath

$MPcond$
$D : \mathbb{N} \nrightarrow \mathbb{N}$

$D' = \{t \mapsto x \mid t \in V \land x = min\{c : \mathbb{N} \mid c = cost\ w\ B \land B \in Apth(E, w, 1, t)\}\}$

$MinPath$ 内の V を変数 T に置き換えたスキーマを $MPApp$ とすれば、$T' = V$ を評価するスキーマ $AssV$ の後に、$MPApp$ を評価することによって、$MinPath$ と同じ結果が得られる。

AssV

$MPcond$
$T' : \mathbb{N}$

$T' = V$

MPApp

$MPcond$
$T : \mathbb{N}$
$D' : \mathbb{N} \nrightarrow \mathbb{N}$

$D' = \{t \mapsto x \mid t \in V \land x = min\{c : \mathbb{N} \mid c = cost\ w\ B \land B \in Apth(E, w, 1, t)\}\}$
$T = V$

$[AssV][MPApp]$ と同じ結果は、以下のスキーマ $MPInit, MPRevStep$ を使って $[MPInit]$ $[MPRevStep]^*$ で表される。

MPInit

$MPcond$
$T' : \mathbb{N}$
$D' : \mathbb{N} \nrightarrow \mathbb{N}$

$T' = \{1\}$
$D' = \{1 \mapsto 0\}$

すなわち、この $MPInit$ の評価後、以下の $MPRevStep$ の反復によって同じ結果が得られる。この $MPRevStep$ が $dcond$ 条件を満たすことも証明する事が出来る。

MPRevStep

$$
\begin{array}{l}
MPcond \\
T, T' : \mathbb{N} \\
D, D' : \mathbb{N} \pfun \mathbb{N} \\
\hline
D = \{t \mapsto x \mid t \in T \land x = min\{c : \mathbb{N} \mid c = cost\ w\ B \land B \in Apth(E, w, 1, t)\}\} \\
T \subseteq V \land T \neq V \Rightarrow (T' = T \cup \{q\} \land \\
\quad q \in V - T \land \\
\quad \exists k \in T : \forall r \in V - T : \\
\quad cost\ q\ k + D\ k \leq min\{c : \mathbb{N} \mid c = cost\ w\ B \land B \in Apth(E, w, 1, r)\} \land \\
\quad D' = D \cup \{q \mapsto (k + D\ k)\})
\end{array}
$$

IO 正則式で表現した洗練化の過程は、以下のようになる。

$$[MinPath] \sqsubseteq [AssV][MPApp]$$

$T = V$ の下での $MPApp$ の結果は $T = \{\}, D = \{1 \mapsto 0\}$ の状態から出発し $MPRevStep$ を順次評価して，$T = V$ に達したときの結果と同じであるから、

$$[AssV][MPApp] \sqsubseteq [MPInit][MPRevStep]^*$$

と表せる *(*厳密な証明は省略*)*。

4.6 IO 正則式に基づく洗練手法の評価

Z 上の IO 正則式に基づく仕様表現による洗練化の方法を述べた。IO 正則式はコンピュータの静的な論理関係の部分と動的な状態推移の部分を分離して表現する記法である。仕様，設計いずれにおける表現としても有効であると考えられる。特に、外部環境との相互作用があるシステムでは、入力出力に伴うシステム状態の推移を，ストリームや列などのデータ構造によって表現することが困難である。このような場合には、IO 正

則式によって仕様を記述することが有効である。データフローネットワークにおけるプロセスは、ストリームをストリームに変換する機構と考えられるが、直接ストリーム間の関係として表現する事が容易ではなく、また、設計への移行がスムーズではないことから、IO 正則式はデータフローネットワークのプロセスに対する表記法として最初に提案されたものであるが、IO 正則式には、並行処理の記述がなく、その点に対する拡張が課題として残されている。

IO 正則式の基本項はZスキーマが適合し、本節ではZに基づいた IO 正則式が仕様，設計の記法として提案され、考察された。

従来の洗練化に対し、優れている点は、

- 洗練化過程も IO 正則式という統一した記法で進めることができる。
- スキーマの複合した形を洗練化の対象とすることができる。
- プログラムにおける制御の条件を最終段階まで腐心する必要がない。

それに対し、洗練化の結果がプログラムではなく IO 正則式であるために、最後にプログラムに変換する過程が必要になり、その為の証明規則がさらに必要になる点が、従来の方法に比べると不満である。変換されたプログラムの IO 正則式に対する正当性の検証については、次節で提案される。

5　IO 正則式に対するプログラム検証

5.1　動的性を盛り込んだ仕様としての IO 正則式

Ｚの表記では, プログラムの各部分に対応する前条件と後条件の関係がスキーマで表現される．しかしそれぞれのスキーマの関係は, スキーマ内の述語部が表す状態による結びつきのみである．したがって, 洗練化によって作り出されるプログラムは, １つの制御プログラムがすべてのイベント発生に対応してハンドラ（すなわちＺのスキーマ）を制御するかのようなものになる．このように, 基本的にＺは静的な表現による仕様記述であり, システムの振る舞いを表現することには十分対応していない．そのために, いくつかの提案がなされている．それらは, Ｚスキーマを包含した動的性を表現する仕様記述と, Ｚスキーマ中に動的性を盛り込む仕様記述の, 大きく２通りの方法に分類することができる．前者として, CSP[18],CCS[30] などとの結合を試みた仕様記述の研究 [13, 12, 34, 37] がある．これらは並行プロセスを含むモデルを対象にしており, 手続きプログラムへの洗練化を目的とするときには, 必ずしも適しているとはいい難い．また, 環境との相互作用があるシステムの振る舞いを状態遷移によって表現する方法の研究 [35] もある．やはり, 状態遷移と目的プログラムとの関係付けが困難である．後者に分類される方法として, 操作や状態のヒストリをＺスキーマに盛り込む方法 [9, 11], 時制論理の記述をスキーマに許した表現 [10] なども提案されている．いずれも, 入出力を伴う個々のプログラムとの対応が明らかではなく, スキーマに新たに振る舞いを盛り込むことは洗練化や検証における複雑さを増すことになる．

以下では, ＺのスキーマをＩＯ正則式 [16] の基本項に対応させ, スキーマの実行順序も含めた仕様表現を提案する．これは, 上の分類では前者に含まれるものであり, この表現によって, 並行処理を含まない手続き的プログラムに対する振る舞いも仕様記述に含まれ, 標準的な基本３構造に基づいたプログラムに仕様が対応することになる．さらに, このＺ上のＩＯ正則式による仕様を基に, ホーア論理 [17, 45] を基礎にしたプログラムの正当性検証も可能であり, 本稿ではその為の規則と検証の例を紹介する．

5.2　Ｚによる仕様の問題点

環境との相互作用（interaction）を伴うシステムでは, どのような順序で入出力やイベント発生が進行するかが問題である。一方，Ｚの一つのスキーマは, 事前事後における状態の関係を記述するのみで, 連続する相互作用を記述する事は容易ではない。

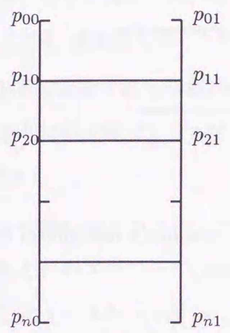


図-1: あみだくじ.

本章では, 簡単な例でＺによる仕様記述の問題点を考察する．

Example 5.1 あみだくじを表示する．水平線を引く位置は乱数によって定めるとする．さらにプレーヤによって選択された端から線をたどりその経路を太線で描く．ただし簡単にするため, 本稿では垂直線が２本だけとしている．

この問題に対して, 以下のようなＺ仕様が自然に得られる．

Points

$p : \mathbb{N} \times \{0,1\} \rightarrow \mathbb{N} \times \mathbb{N}$　– p_{ij} *is a point of* $i-th$ *row and* $j-th$ *collum.*

$n : \mathbb{N}$　– *the number of rows.*

$n = 10$

Init

$i' : \mathbb{N}$

$i' = 0$　– *Initialize* i *as the horizontal line number* 0.

Vertical

Points

$line! : \mathbb{P}((\mathbb{N} \times \mathbb{N}) \times (\mathbb{N} \times \mathbb{N}))$　– $line!$ *is a set of lines to be drawn.*

$line! = \{(p_{00}, p_{n0}), (p_{01}, p_{n1})\}$　– *Draw two vertical lines.*

Choice

$i', k' : \mathbb{N}$

$sel? : \mathbb{P}(\mathbb{N})$ — *sel? is a set whose element is a selected column number.*

$i' = 0$

$\{k'\} = sel?$

Horizontal

Points

$i, i' : \mathbb{N}$

$line! : \mathbb{P}((\mathbb{N} \times \mathbb{N}) \times (\mathbb{N} \times \mathbb{N}))$

$bp' : \mathbb{N} \times \{0,1\} \pfun \{0,1\}$ — $bp_{ij} = 1$ *means* $i-th$ *row has a horizontal bar.*

$i \leq n$

$i' = i + 1$

$(rand() = 1 \land bp'_{i0} = 1 \land line! = \{(p_{i0}, p_{i1})\})$

$\lor (rand() = 0 \land bp'_{i0} = 0)$

Trace

Points

$i, i' : \mathbb{N}$

$bline! : \mathbb{P}((\mathbb{N} \times \mathbb{N}) \times (\mathbb{N} \times \mathbb{N}))$ — *bline! is a set of lines to be traced.*

$bp : \mathbb{N} \times \{0,1\} \pfun \{0,1\} \Delta points$

$i \leq n$

$i' = i + 1$

$(bp_{i0} = 0 \land bline! = \{(p_{ik}, p_{i'k})\})$

$\lor (bp_{i0} = 1 \land bline! = \{(p_{ik}, p_{i(k \oplus 1)}), (p_{i(k \oplus 1)}, p_{i'(k \oplus 1)})\})$

*line!*と*bline!*はそれぞれ, 与えられた２点間を細線と太線で引くことを表す. 関数$rand()$は二値{0,1}の乱数を求める関数を意味する. 演算子$\oplus$は２を法とする和を意味する.

この仕様は以下のように実行されるプログラムを意図している. スキーマ *Init* によって初期値設定がなされ, 次に *Vertical* の部分が実行され垂直線が引かれる. さらに, *Repeat*

が繰り返されて, 幾つかの水平線が引かれる. その後,*Choice* によって, ２つの垂直線のいずれかがプレーヤによって選ばれ,*Trace* によって, 選ばれた方の辿る経路が太線で表示される.

スキーマ *Init* が最初に実行されることを除けば, いずれのスキーマも, それが実行される為の条件を含んではいない. 個々のスキーマの実行条件を含める為には, 状態変数を新たに加え, 垂直線を引くための状態, 水平線を繰り返し引くための状態, 選択を入力する為の状態などをそれらの変数によって表現する必要がある. そのことで, 仕様は複雑になり, しかも仕様段階ですでに明らかな構造（連接や反復など）が直接的に表現されるわけではない.

この例におけるスキーマに実行順序を表現する変数 turn を盛り込むことを試みる。それぞれのスキーマＳの述語部に論理式ｐを連言によって加え，それに伴う宣言も宣言部に加えてスキーマを $[S \land \{p\}]$ で表し，そのスキーマに対応するプログラムの実行を $compute[S \land \{p\}]$ で表すことにする。このとき、全体のプログラムを Pascal の case 文を使って表現すれば、以下のようになる。

```
turn := 0;
while turn <  5  do
  case turn of
    0: compute [Init ∧  {turn=0 ∧ turn'=1} ] ;
    1: compute [Vertical  ∧  {turn=1 ∧ turn'=2} ] ;
    2: compute [Choice  ∧  {turn=2 ∧ turn'=3} ] ;
    3: compute [Horizontal ∧ {i<n ∧ turn=3 ∨ i=n ∧ turn=3 ∧ turn'=4}] ;
    4: compute [Trace ∧  {i<n  ∧ turn=4 ∨ i=n ∧ turn=4 ∧ turn'=5}]
  end;
```

このプログラムには、実行順序制御のために turn という変数が使われている。さらに、*Horizontal*, *Trace* の部分は反復されるものであるが、このプログラムには反復構造は見

られない。すなわち、構造化されたプログラムとは言いがたい。

スキーマの実行順序, 反復構造, 選択構造などをスキーマを組み合わせた式で表現することによって, 仕様が解りやすくなるとともに, 手続き的プログラムへの洗練化の助けにもなる.

5.3 While プログラム

本稿では, 目的のプログラミング言語としては, 以下の基本的構文を持つ言語を扱う.

$$
\begin{aligned}
< statement >::= \quad & < empty > \\
& |< variable >:=< expression > \\
& | read(< input\ channel >, < variable >) \\
& | write(< output\ channel >, < expression >) \\
& | begin < statement\ sequence > end \\
& | if < logical\ expression > then < statement > \\
& \quad else < statement > \\
& | while < logical\ expression > do < statement > \\
< statement\ sequence >::= \quad & < statement > \\
& |< statement >; < statement\ sequence > \\
< program >::= \quad & < statement >
\end{aligned}
$$

以下では,$< statement >$を文,$< statement\ sequence >$を文の列と呼ぶ. $read(IN, x)$ はチャネル IN から入力した値を表し, 同時に変数 x にその値が代入される. 同様に, $write(OUT, e)$ はチャネル OUT に式 e の値を出力することを意味する. さらに式としての $read(IN, x)$ は,if 文や while 文の条件式$< logical\ expression >$に 1 度だけ現れることが可能であるとする.

このプログラミング言語の部分正当性に対する,Hoare 論理の公理 [45] は以下のようになる. 推論規則は省略するが, 以降 $\{P\}S\{Q\}$ によって, 部分正当性を表すものとする.

Axiom 1 *Hoare* 論理の公理

1. 代入文の公理 : $\{A[\![t/x]\!]\}x := t\{A\}$
2. 空文の公理 : $\{A\}\{A\}$

3. *read* 文の公理 : $\{A[\![rI \cup \{x\}/rI]\!]\}read(I, x)\{A\}$
4. *write* 文の公理 : $\{A[\![oO \cup \{e\}/oO]\!]\}write(O, e)\{A\}$

なお,$\{A[\![t/x]\!]\}$ は A の中の変数 x を式 t で置き換えることを意味する。上の公理で, 表明中の I はチャネル I から入力されたデータ集合の列 (sequence) を表し, rI はチャネル名 I に連動した名前で, 表明 $\{A\}$ 内でチャネル I から入力されるデータ集合を表す. 同様に,oO はチャネル名 O に連動した名前で, 表明 $\{A\}$ 内でチャネル O に出力されるデータ集合を表す.

read 式を含む while 文や if 文は次のように,read 動作の後に入力列への追加が行われるものと解釈する.

Definition 5.1 *while* 文 *(while w do S)* が *read* 式を w に含むなら, 推論規則はその *while* 文を以下のプログラムに変換してから適用する.

$$
\begin{aligned}
& while\ \ w[\![x/read(I, x)]\!]\ do \\
& \quad begin\ \ I := I^\frown\{x\};\ \ S\ end; \\
& I := I^\frown\{x\};
\end{aligned}
$$

if 文 *(if w then S1 else S2)* が *read* 式を w に含むなら, 推論規則はその *if* 文を以下のプログラムに変換してから適用する. *else* 以下がない *if* 文に対しても同様とする.

$$
\begin{aligned}
& if\ \ w[\![x/read(I, x)]\!]\ then \quad begin\ \ I := I^\frown\{x\};\ \ S1\ end \\
& else \qquad\qquad\qquad\qquad\quad begin\ \ I := I^\frown\{x\};\ \ S2\ end;
\end{aligned}
$$

$I^\frown x$ は列 I の最後に x を追加した列を意味する. 入力変数, 出力変数は, 表明においては現時点で入力, 出力されたデータ (すなわち列の最後尾) を意味する. それを $last(IN)$ と表す. $last(IN^\frown x) = x$ である. 入力列, 出力列の名前はチャネル名と同じものとして扱う. 従って,$read(IN, x)$ なる入力に対しては, スキーマ内の $x?$は $last(IN)$ として扱い,$write(OUT, w)$ なる出力に対しては, スキーマ内の $w!$は $last(OUT)$ として扱う.

Definition 5.2 S を *read* 文 *(read(I, x))* を含む文の列とする. このとき,$\tilde{S}$ は以下の文の列を意味するものとする. なお,ϕ は空集合を表す.

$$iI := \phi;\ S;\ I := I^\frown iI.$$

S を *write* 文 $(write(O,x))$ を含む文の列とする．このとき,$\tilde{S}$ は以下の文の列を意味するものとする．

$$oO := \phi;\ S;\ O := O^\frown oO.$$

もし文の列 S に入力, 出力文が含まれないとき, $\tilde{S} = S$　である．

5.4　ＩＯ正則式に基づいた証明の規則

$[\,\alpha\,]$ によってスキーマαの述語部を表すものとし, これを基本項 (*primary term*) と呼ぶ．スキーマの宣言部を, 本稿ではほとんど無視して扱うことになる．一方,Hoare 論理による表明は $\{\ \}$ によって括って表す．表明とスキーマの大きな相異点は, スキーマが変数として, 代入実行前後の値を表現する変数（v'と v の如く）を含むことができる点と, 入力, 出力を表す変数（$in?$や $out!$など）を含むことができる点である．Ｚに基づいたＩＯ正則式は以下の定義で与えられる (再掲)．

Definition 5.3

$$\begin{aligned}
\langle factor\rangle &::= \langle primary\,term\rangle \mid [\,\langle term\rangle\,] \\
&\quad \mid \langle primary\,term\rangle^* \mid [\,\langle term\rangle\,]^* \\
\langle seq\ factor\rangle &::= \langle\ factor\rangle \mid \langle seq\ factor\rangle\langle\ factor\rangle \\
\langle term\rangle &::= \langle seq\ factor\rangle \mid \langle term\rangle \sqcup \langle seq\ factor\rangle \\
\langle IO\ regular\ expression\rangle &::= \langle term\rangle
\end{aligned}$$

ただし、$\langle seq\ factor\rangle, \langle term\rangle$ の右辺の演算は左結合とする。

Definition 5.4 S を文の列, p を論理式, さらに $[\alpha]$ を述語部が q であるスキーマとする．v は S に現れる変数 $v_1, v_2, \ldots, v_k$ を表し、R は一時的に置き換えるための変数 $R_1, R_2, \ldots, R_k$ で、S, q に現れない変数を表すものとする。このとき、$W = p \wedge R_1 = v_1 \wedge R_2 = v_2 \wedge \ldots \wedge R_k = v_k$ と置くとき、*Hoare* 論理の部分正当性の３個組 *(Hoare's triple)* として

$$W\ \ \tilde{S}[\![R/v]\!]\ \ q[\![R/v', last(I)/i?, last(O)/o?]\!]$$

が成り立つとき、

$$\{p\}\ S\ [\alpha]$$

と表す．式のなかで、a/b は b を a で置き換えることを意味する。$i?, o!$は入力変数全体, 出力変数全体を表し、I, O は入力、出力のための列の全体を表す。

この Definition5.4 に対する例を上げる。次は、入出力が含まれていないプログラムに対する例である。

Example 5.2 $p \equiv x + y = 10, S \equiv x := x + y;\ y := x * 2$ とし、$[\alpha]$ の述語部 $q \equiv y' = x' + 10$ とする。プライム付きで現れる x, y をそれぞれ、$r1, r2$ で一時的に置き換えると、上の定義における　$\tilde{S}[\![R/v]\!], q[\![R/v']\!], W$ はそれぞれ、

$$\begin{aligned}
\tilde{S}[\![R/v]\!] &\equiv r1 := r1 + r2;\ r2 := r1 * 2 \\
q[\![R/v']\!] &\equiv r2 = r1 + 10 \\
W &\equiv x + y = 10 \wedge r1 = x \wedge r2 = y
\end{aligned}$$

に対応する。*Hoare* 論理として

$$W\ \ \tilde{S}[\![R/v]\!]\ \ q[\![R/v']\!]$$

の関係が成り立つので、

$$\{p\}\ \ S\ \ [\alpha]$$

が成り立つことになる。

プログラムやスキーマに入出力が含まれている場合の例を上げる。

Example 5.3 スキーマ $[\alpha]$ の述語部 q が $q \equiv IN? = \{x\} \wedge y' = x + 10$ であるとすれば、$IN?$は $last(IN)$ に、y'は適当な一時変数 r に置き換えられて

$$q[\![r/y', last(IN)/IN?]\!] \equiv last(IN) = \{x\} \wedge y' = x + 10$$

を意味する。

文の列 S を $S \equiv read(IN, x);\ y := x + 10$　とすれば、

$$\tilde{S}[\![r/y]\!] \equiv iIN := \phi;\ read(IN, x);\ r := x + 10;\ IN := IN^\frown iIN$$

となる。$q[\![r/y', last(IN)/IN?]\!]$ を出力条件，$\{true\}$ を入力条件として、出力条件から前方向に順次事前条件である表明を求め、記述すると、以下のようになる。

$$
\begin{array}{l}
\{true\} \\
\{\phi \cup \{x\} = \{x\}\} \\
\quad iIN := \phi; \\
\{iIN \cup \{x\} = \{x\}\} \\
\quad read(IN, x); \\
\{iIN = \{x\} \wedge x + 10 = x + 10\} \\
\quad r := x + 10; \\
\{iIN = \{x\} \wedge r = x + 10\} \\
\quad IN := IN^\frown iIN \\
\{last(IN) = \{x\} \wedge r = x + 10\}
\end{array}
$$

こうして、

$$\{true\}\ \ S\ \ [\alpha]$$

が成り立つ。

以下は，ＩＯ正則式で表現された仕様に対するプログラム検証の規則である．

Definition 5.5 以下で $[\alpha], [\alpha 1], [\alpha 2]$ などは IO 正則式とする．

1. 条件文の規則：$S1, S2$ を文とし,w が $read(I, x)$ を含まないとする．

$$\frac{\{p \wedge w\}S1[\alpha 1]\ \ \{p \wedge \neg w\}S2[\alpha 2]}{\{p\}if\ w\ then\ S1\ else\ S2[\alpha 1] \sqcup [\alpha 2]}$$

2. 連接の規則：$S1$ を文,$S2$ を文の列とする．

$$\frac{\{p\}S1[\alpha 1]\ \ \{q\}S2[\alpha 2]\ \ \{p\}\tilde{S1}\{q\}}{\{p\}S1;\ \ S2[\alpha 1][\alpha 2]}$$

3. 複合文の規則：S を文の列とする．

$$\frac{\{p\}S[\alpha]}{\{p\}begin\ S\ end[\alpha]}$$

4. while 文の規則：S を文とし，w が $read(I, x)$ を含まないとする．

$$\frac{\{p \wedge w\}S[\alpha]\ \ \{p \wedge w\}\tilde{S}\{p\}}{\{p\}while\ w\ do\ S\ [\alpha]^*}$$

この定義を基に連接の規則に関する結合性を示すことができる。

Lemma 5.6 $S1, S2, S3$ を任意の文の列，p を任意の論理式とする。$q = sp(p, S1)$ かつ $\{q\}\ S2\ \{r\}$ である論理式 p, q, r に対して $\{p\}S1[\alpha 1]$, $\{q\}S2[\alpha 2]$, $\{r\}S3[\alpha 3]$ が成り立つとする。このとき、以下の２つの関係は等価である。

$$\{p\}(S1;\ \ S2);\ \ S3[\alpha 1][\alpha 2][\alpha 3] \tag{21}$$

$$\{p\}S1;\ \ (S2;\ \ S3)[\alpha 1][\alpha 2][\alpha 3] \tag{22}$$

proof: 式 *21* を仮定する。このとき、$\{p\}S1;\ S2[\alpha 1][\alpha 2]$ であるから、$\{q\}S2[\alpha 2]$ が成り立っている。これと、与えられた条件 $\{q\}\ S2\ \{r\}$,$\{r\}S3[\alpha 3]$ から、

$$\{q\}S2;\ S3[\alpha 2][\alpha 3]$$

が成り立つ。$q = sp(p, S1)$ であることから、

$$\{p\}S1\{q\}$$

であり、さらに

$$\{p\}S1[\alpha 1]$$

が与えられているので、式 *22* が成り立つ。逆も同様に示される。

この lemma の性質によって、複数の文の並びからなる連接の証明に関しては、規則を適用する順序は限定されないことになる。

Example 5.4 条件文の規則の適用例：条件文を

$$S \equiv if\ x > 0\ then\ y := x\ else\ y := 0$$

とし、スキーマを

$$[x > 0 \wedge y' = x] \sqcup [x \leq 0 \wedge y' = 0]$$

とする。このとき、$\{true \wedge x > 0\}\ y := x\ [x > 0 \wedge y' = x]$ と $\{true \wedge x \leq 0\}\ y := 0\ [x \leq 0 \wedge y' = 0]$ が共に成り立つ。従って、

$$\{true\}\ \ S\ \ [x > 0 \wedge y' = x] \sqcup [x \leq 0 \wedge y' = 0]$$

が成り立つ。

条件文を変えずに、スキーマを

$$[y' = x] \sqcup [y' = 0]$$

とした場合でも、$\{true \wedge x > 0\}\ y := x\ [y' = x]$ と $\{true \wedge x \leq 0\}\ y := 0\ [y' = 0]$ が共に成り立つので、

$$\{true\} S [y' = x] \sqcup [y' = 0]$$

が成り立つ。

この例のように、スキーマに含まれる条件の与え方によって、それを満たすプログラムに幅が生ずるが、その為にスキーマによる仕様が，プログラムを決定するに十分な条件を含むようあらかじめ作られていることが必要である。

Example 5.5 *while* 文の規則の適用例：*while* 文を

$$S \equiv\ while\ x > 0\ do\ \ begin\ tt := tt + x;\ \ x := x - 1\ end$$

とし、スキーマ $[\alpha]$ を

$$[\alpha] \equiv\ [x > 0 \wedge tt' = tt + x \wedge x' = x - 1]$$

とする。このとき、$\{true \wedge x > 0\}\ begin\ tt := tt + x;\ \ x := x - 1\ end\ [x > 0 \wedge tt' = tt + x \wedge x' = x - 1]$ と $\{true \wedge x > 0\}\ begin\ tt := tt + x;\ \ x := x - 1\ end\ \{true\}$ が満たされるので、

$$\{true\}\ \ S\ \ [\alpha]^*$$

が成り立つ。

なお、スキーマ $[\beta]$ が

$$[\beta] \equiv\ [x' = x - 1]$$

の場合でも，$\{true \wedge x > 0\} begin\ tt := tt + x;\ \ x := x - 1\ end [x' = x - 1]$ が満たされるので、やはり、

$$\{true\}\ \ S\ \ [\beta]^*$$

が成り立つ。

この例に示されるように、反復のIO正則式が停止しないことがある場合でも規則を満たすwhileループのプログラムが停止することはあり得る。逆にIO正則式が停止するのであれば、while文の規則から得られるプログラムは，停止することが保証される。すなわち、ＩＯ正則式の仕様が停止性を満たしていれば、その仕様を満たすプログラムは停止性を満たすことになる。

5.5 証明の例

Example 5.6 はじめに掲げたあみだを求める例に対し，プログラムの検証を行う．ＩＯ正則式による仕様は

$[Init]\,[Vertical]\,[Horizontal]^*\,[Choice]\,[Trace]^*$

である．以下のプログラムがこの仕様を満たすことを証明する．

```
program ladder();
  type pair=record  x,y:integer
            end;
  const n=10;
  var i,n,k:integer;
      p:array[0..1,0..n] of pair;
      bp:array[0..1,0..n] of (0,1);
begin
  i:=0;                                  } S1
```

```
write(LINE,(p[0,0],p[n,0]));
write(LINE,(p[0,1],p[n,1]));

while i<= n  do
  begin
    if rand()=1 then
       begin
         bp[i,0]:=1;
         write(LINE,(p[i,0],p[i,1]))
       end
    else
       bp[i,0]:=0;
    i:=i+1
  end;

i:=0;
read(SEL,k);

while i<= n  do
  begin
    if bp[I,0]=1 then
       begin write(BLINE,(p[i,k],p[i,mod(k+1)]));
             write(BLINE,(p[i,mod(k+1)],p[i+1,mod(k+1)]))
       end
    else
       write(BLINE,(p[i,k],p[i+1,k]));
    i:=i+1
  end;
end.
```

1. 基本項の公理から, $\{true\}$ i:=0 $[i' = 0]$.

   従って, $\{true\}S1[Init]$ が成り立つ.

2. $S2$ の部分の証明：

$S2$ は次のプログラムに変換される.

oLINE $= \phi$;

write(LINE, (p[0, 0], p[n, 0]));

write(LINE, (p[0, 1], p[n, 1]));

LINE = LINE$^\frown$oLINE.

以下は, 個々の文と事後条件に対する事前条件を表す.

$\{true\}$ oLINE $= \phi$

$\{oLINE = \phi\}$ write(LINE, (p[0, 0], p[n, 0]))

$\{oLINE = \{(p_{00}, p_{n0})\}\}$ write(LINE, (p[0, 1], p[n, 1]))

$\{oLINE = \{(p_{00}, p_{n0}), (p_{01}, p_{n1})\}\}$ LINE = LINE$^\frown$oLINE

$\{last(LINE) = \{(p_{00}, p_{n0}), (p_{01}, p_{n1})\}\}$

よって, $\{true\}$ $S2$ $[Vertical]$ が成り立つ.

3. *while* 文の規則を $S3$ に適用する. 前提は以下の２つである.

   １つは, $\{i \leq n\}$ $S31$ $[Horizontal]$ もう１つは, $\{i \leq n\}$ $S31$ $\{true\}$.

   この２つの前提から, 次の結論が導かれる.

   $\{true\}$ $S3$ $[Horizontal]^*$

4. 基本項の公理から,

   $\{true\}$i := 0; read(SEL, k) $\left[\ i' = 0 \wedge \{k'\} = sel?\ \right]$.

   故に $\{true\}$ $S4$ $[Choice]$ が成り立つ.

5. $S5$ に *while* 文の規則を適用すると, ステップ *3* と同様にして, $\{true\}$ $S5$ $[Trace]^*$ が証明される.

   PAD 図で表現すると下図のように表すことができる。

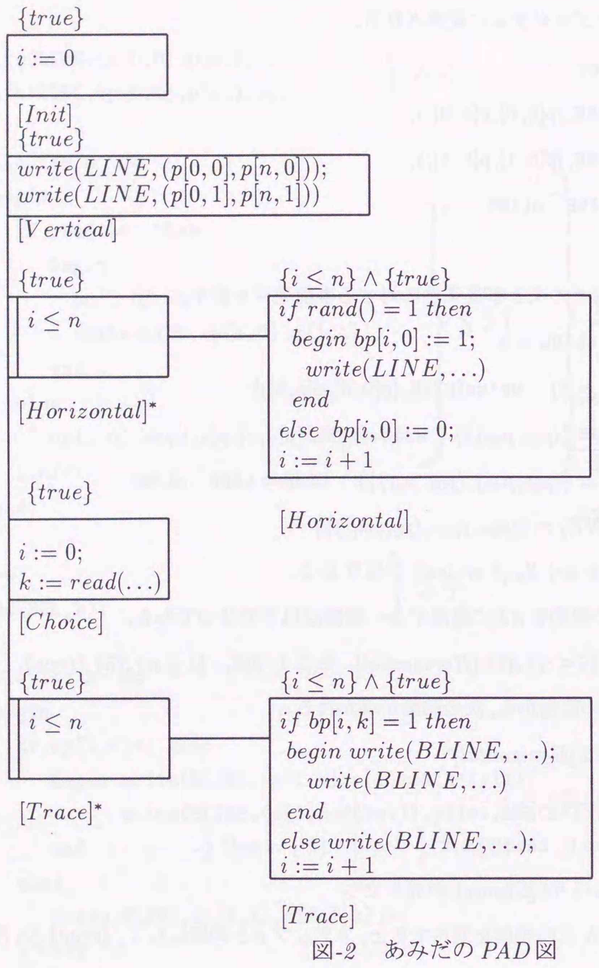


図-2　あみだの *PAD* 図

6. 文の列の規則から, 以下が成り立つ.

$$\frac{\{true\}S1\,[Init] \quad \{true\}S2[Vertical] \quad \{true\}\tilde{S}1\{true\}}{\{true\}S1;\ S2\,[Init]\,[Vertical]}.$$

$$\frac{\{true\}S1;\ S2\,[Init]\,[Vertical] \quad \{true\}S3\,[Horizontal]^* \quad \{true\}\tilde{S}1;\ \tilde{S}2\{true\}}{\{true\}S1;\ S2;\ S3\,[Init]\,[Vertical]\,[Horizontal]^*}.$$

$$\frac{\begin{array}{l}\{true\}S1;\ S2;\ S3\,[Init]\,[Vertical]\,[Horizontal]^* \\ \{true\}S4\,[Choice] \\ \{true\}\tilde{S}1;\ \tilde{S}2;\ \tilde{S}3\{true\}\end{array}}{\{true\}S1;\ S2;\ S3;\ S4\,[Init]\,[Vertical]\,[Horizontal]^*\,[Choice]}.$$

$$\frac{\begin{array}{l}\{true\}S1;\ S2;\ S3;\ S4\,[Init]\,[Vertical]\,[Horizontal]^*\,[Choice] \\ \{true\}S5\,[Trace]^* \\ \{true\}\tilde{S}1;\ \tilde{S}2;\ \tilde{S}3;\ \tilde{S}4\{true\}\end{array}}{\{true\}S1;\ S2;\ S3;\ S4;\ S5\,[Init]\,[Vertical]\,[Horizontal]^*\,[Choice]\,[Trace]^*}.$$

これが求める結果である.

Example 5.7 自然数の数値を複数入力し、その平均値を求めて最後に出力するプログラムを考える。入力のストッパは値 0 とする。

この問題に対し一つの *Z* のスキーマで仕様を記述することは可能である。しかし、仕様からの詳細化 *(refinement)* あるいは作成されたプログラムに対する検証 *(verification)* などを考えると，ある程度の小さなスキーマに分解しておく方が有利である。そのような立場で考えると，以下のようなZ仕様が自然に得られる。

Reg

$$sum, numb : \mathbb{N}$$

Init

$$\Delta Reg$$

$$sum' = 0$$
$$numb' = 0$$

Repeat

ΔReg

$in? : \mathbb{P}\mathbb{N};\ kg : \mathbb{N}$

$in? = \{kg\} \land x > 0$

$sum' = sum + kg$

$numb' = numb + 1$

Last

ΔReg

$in? : \mathbb{P}\mathbb{N}$

$out! : \mathbb{P}\mathbb{N}$

$in? = \{0\}$

$out! = \{sum/numb\}$

実は以下のプログラムもこの仕様を満足する。実際、詳細化によって得られるプログラムはこのような構造のものになるはずである。しかし、このプログラムでは、入力値0に対し、出力を出すだけで終了はしない。

```
begin
  sum:=0;
  numb:=0;
  while  read(IN,kg)>= 0  do
      if kg>0 then
        begin
          sum:=sum+kg;
          numb:=numb+1
        end
      else
```



```
        write(OUT,sum/numb);
end
```

すなわち、作成したZ仕様を完全なものにするためには、新たに状態変数を設け、平均値の出力が実行されるべき状態や、プログラム実行終了の状態を，状態変数で表現するように変更をしなければならない。しかし、そのような不自然な状態変数を設けることは、仕様記述をより複雑にすることになる。

ＩＯ正則式による仕様は

$$[Init][Repeat]^*[Last]$$

である。

以下のプログラムがこのIO正則式を満たすことを証明する。便宜上プログラムの各部分に名前を付けて扱う。以下を$S1$

```
sum:=0;
numb:=0
```

以下を$S2$

```
while  read(IN,kg)>0  do
  begin
    sum:=sum+kg;
    numb:=numb+1
  end
```

以下を$S3$

```
  write(OUT,sum/numb)
```

とそれぞれ置き、$S1$; $S2$; $S3$ がこの仕様を満たすことを示す。

1. 基本項の規則より、

$\{true\}$

```
sum:=0; numb:=0
```

$[sum' = 0 \wedge numb' = 0]$

2. *while* 文の規則 *(read* 式を含む場合*)* の前提は以下になる。

$\{kg > 0\}$

```
IN:=IN⌒{kg} ; sum:=sum+kg ; numb:=numb+1
```

$[last(IN) = \{kg\} \wedge kg > 0 \wedge sum' = sum + kg \wedge numb' = numb + 1]$

および

$\{kg > 0\}$

```
IN:=IN⌒{kg} ; sum:=sum+kg ; numb:=numb+1
```

$\{true\}$

これら前提より、*while* 文の規則の結論として以下が得られる。ただし、以下の *while* 文は *read* 式を消去した後の *while* 文（これを$\hat{S2}$と置く）に変換している。

$\{true\}$

```
while  kg>0  do
  begin
    IN:=IN⌒{kg};
    sum:=sum+kg;
    numb:=numb+1
  end
```


$[last(IN) > 0 \wedge sum' = sum + kg \wedge numb' = numb + 1]^*$

3. *while* 文の終了時の処理を $S3$ に加えた文の列（これを$\hat{S3}$と置く）に対し *write* 文の規則より、

$\{kg = 0\}$

```
oOUT:={};
IN := IN⌒{kg};
write(OUT,sum/numb);
OUT:=OUT⌒oOUT
```

$[last(IN) = \{0\} \wedge last(OUT) = \{sum/numb\}]$

4. 連接の規則より、

$$\frac{\{true\}S1[Init] \quad \{true\}S2[Repeat]^* \quad \{true\}\hat{S1}\{true\}}{\{true\}S1;\ S2[Init][Repeat]^*}$$

$$\frac{\{true\}S1;\ S2[Init][Repeat]^* \quad \{last(IN) = \{0\}\}S3[Last] \quad \{true\}\hat{S1};\ \hat{S2}\{last(IN) = \{0\}\}}{\{true\}S1;\ S2;\ S3[Init][Repeat]^*[Last]}$$

5. 複合文の規則より、

$$\frac{\{true\}S1;\ S2;\ S3[Init][Repeat]^*[Last]}{\{true\}begin\ S1;\ S2;\ S3\ end[Init][Repeat]^*[Last]}$$

よって証明された。

なお、証明における規則の適用関係は，ＰＡＤ図（図-*3*）で示すと解りやすい。

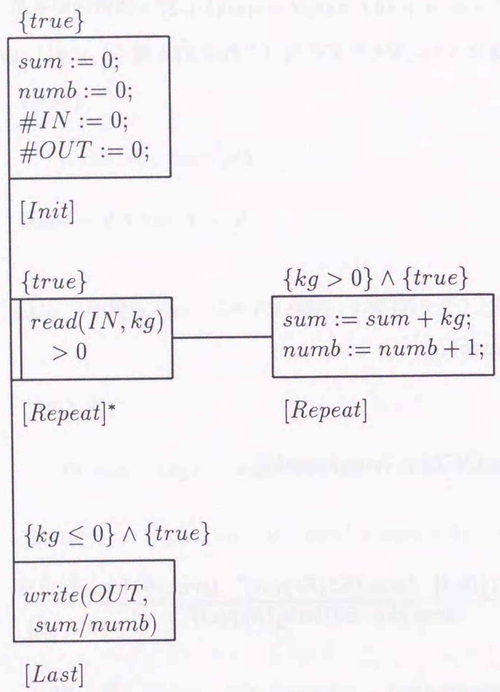


図-3:平均値計算の PAD 図.

5.6 IO 正則式に対するプログラム検証の評価

Zを基にしたIO 正則表現によって仕様を表現し, プログラムの検証を行うことを提案した．IO 正則式による仕様化は, 外部との相互作用に伴って状態が変化するシステムに対し有効である．しかしIO 正則式は並行動作を含むモデルではない．その点でCSP やCCS によるモデルと異なっているが, 手続き的プログラムを対象のプログラムとした場合には,IO 正則式がむしろ洗練化に適していると考えられる．

本論文ではIO 正則式への意味の付与を, 部分正当性を基にして行っている．すなわち, 仕様に停止性を表現することを含んでいない．プログラムの停止性検証は, 最後のプログラム検証段階ではなく,IO 正則式を含む仕様に対して行うことになる．前節のIO 正則式における反復の導入には、停止性を条件としていた（完全正当性）ので、停止性は仕様に盛り込まれているとして扱ってよい。

本節での規則は完全なものではない。すなわち、正当であるプログラムの全体ではなく、その一部分のみがこの規則を用いて証明されるに過ぎない。このことは、規則を念頭において仕様からプログラミングを行う立場においてはさほどの問題ではないと思われる。しかし、本節の規則では, 選択，反復などに対するプログラムの文が限定されている．より現実のプログラムに対応させる為に,for 文やrepeat 文を初めとする, プログラム言語の持つ種々の表現への対応も考えていかねばならない．

6　データフローネットワークの分析

6.1　データフローネットワークによるモデル

データフローネットワーク[25, 22]は、並列に動作する複数のプロセスをFIFOの機構をもつチャネルを介して結合した計算モデルである。本節では、決定性データフローネットワークを形式化し、その表示的意味 (denotational semantics) としてのヒストリ関数の不動点の存在や、不動点と操作的意味 (operational semantics) との対応を考察する。

データフローネットワークの表示的意味を最初に考察したのは、Kahn[22]である。Kahnは、決定性のデータフローネットワークに対し、ヒストリ間の連立方程式を導き、各プロセスが完備な半順序集合上の連続関数を定義するなら不動点定理が適用できることを示した。彼の研究のあと、決定性よりもむしろ非決定性のデータフローネットワークに対する意味論[3, 25, 29]の研究が中心になってきている。Kahnは、決定性データフローネットワークを、簡単な言語で書かれたプログラムの例で説明をしており、ネットワークの厳密な形式化を行なっているわけではない。したがって、例えばどのようなプロセスが連続関数になるか、また、不動点がネットワークのどのような実行結果に対応しているのか、すなわち操作的意味との関係については、何も言及していない。現在までのところ、決定性データフローネットワークのこのような問題に対する他の研究者による報告もない。

本節では、まず (1) 決定性データフローネットワークを形式化した。本論文の形式化では、ネットワークは複数の基本プロセスと、基本プロセスを結ぶチャネルというFIFOデータ伝達機構からなる。全体の構成はKok[26]などと同様であるが、我々は基本プロセスに決定性を保つための制限を設けている。すなわち基本プロセスは、状態保時の有限（または可算無限）メモリと計算可能な部分関数から成る機械であり、複数のチャネルから同時にデータを入力し複数のチャネルに同時に出力を出すが、データを入力するチャネルの組合わせは常に一通りが選択される、したがって決定性の動作をするものである。

つぎに、(2) 基本プロセスがヒストリ（プロセスが処理した文字の並び）の連続関数を定義することを示し、その上で、決定性データフローネットワークが常に入出力ヒストリ間の連続関数を定義することを示している。したがって、任意の決定性データフローネットワークのヒストリ関数は最小不動点を持つことになる。

さらに (3) 決定性データフローネットワークのヒストリ関数の不動点が有限長の場合について、操作的意味の上でどのようなものに対応しているかを考察している。その結果は、次のようにいうことができる。ネットワークの実行中に、すべてのチャネルが空の状態に達した場合、あるいは、すべてのプロセスでチャネルからの入力動作が不能である状態に達した場合は、その段階までの出力ヒストリがネットワークの表すヒストリ関数の不動点である。すなわち不動点の存在は、ネットワークの実行が無限に続くのでなければ、途中でこの２つのいづれかの状態が必ず生ずるということを述べていると解釈することができる。

以下、第２項では、データフローの意味を扱う上での基本となる考えである不動点を紹介する。第３項では種々の基本的記法の定義を与え、第４項では、基本プロセスを定義し、表示的意味であるヒストリ関数を操作的意味によって定義する。第５項は、データフローネットワークを基本プロセスのもとで定義し、ネットワークのヒストリ関数が完備な半順序集合上の連続関数であること、すなわち、最小不動点が存在することを示している。第６項ではネットワークの実行例をあげ、第７章では、最小不動点と操作的意味とを対応づけしている。

6.2　データフローネットワークにおける不動点

データフローネットワークは，プロセス間でチャネルを通して非同期にデータの送受信を行う機構を持つ。すなわち、チャネルはデータのバッファであり、基本的にはその容量に限界はない。プロセスは入力チャネルからの入力データを基に計算をして出力データに変換し出力側のチャネルに送出する。このようなネットワークで、一般には、特定のチャネル上にどのようなデータ列が得られるかが対象とする問題である。なお、以下では、データ列をストリーム (stream) と呼ぶ。例えば、下図のようなデータフローネットワークを考える。プロセスf,g,h0,h1とチャネルY,Z,T1,T2から成る。実行開始の時点

で、h0 が 0 をチャネルYに、h1 が 1 をチャネルZに出力する。その後は、すべてのプロセスは入力すべきデータがチャネルに現れると処理を実行する。すなわち、プロセスfは，チャネルYからのデータをチャネルXに送出し、次にチャネルZからのデータをチャネルXに送出する。この処理を反復する。プロセスgは、チャネルXからのデータに対し、0はチャネルT1に送出し、1はチャネルT2に送出する。プロセスh0，h1は入力データをそのまま、出力チャネルに送出し続ける。このデータフローネットワークは、永久に動作を続ける。そして、チャネルXには、0101...なる、0と1からなるストリームが生ずることになる。

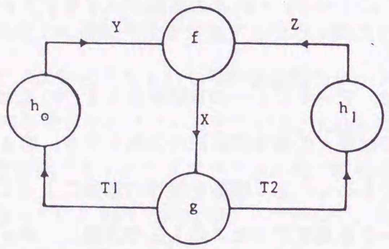


図 1: データフローネットワークの例

このデータフローネットワークは、以下の連立方程式で表現することができる。

$$\begin{aligned} Y &= h0(T1) \\ Z &= h1(T2) \\ X &= f(Y,Z) \\ T1 &= g_1(X) \\ T2 &= g_2(X) \end{aligned}$$

この方程式において、プロセスf,g,h0,h1はストリームをストリームに変換する関数を、チャネルY,Z,T1,T2はそれぞれのチャネル上に生ずるストリームを意味している。

いま D を任意の高々可算無限の集合とする。D^ω によって、集合Dの要素の有限または無限の列全体から成る集合を意味することとする。ただし、D^ω は空列 (Λ) も含むものとする。以下、X,Y は集合 D^ω の任意の要素とする。

定義 6.1 X が　Y の接頭語であるとき、$X \subseteq Y$ と表す。

上の定義により、D^ω は半順序集合になる。$\forall X \in D^\omega : \Lambda \subseteq X$ が成り立つので、最小元 (least element) は、Λである。D^ω の元の列

$$X_1 \subseteq X_2 \ldots \subseteq X_i \ldots$$

の上限を $limi \to \infty X_i$ であらわすと、D^ω は完備な半順序集合 (cpo:complete partial ordered set) になる。コンピュータで処理するデータは文字の列であり、そのようなデータ全体はcpoをなすと考えることができる。

このcpoの上での連続関数は以下で定義される。

定義 6.2 *cpo* 集合をC, Rとし、Cの任意の元の列 $X_1 \subseteq X_2 \ldots \subseteq X_i \ldots$ に対し、C上の関数 $f : C \to R$ が,

$$f(\lim_{i\to\infty} X_i) = \lim_{i\to\infty} f(X_i)$$

を満たすとき、fは連続関数であるという。

プログラムが表す領域上の関数は、データの長さが無限に伸びた場合を想定することができるので、連続関数と見なすのが自然である。

cpo上の連続関数については、以下の定理が成り立つ。

定理 6.1 *C*を *cpo*, fをCからCへの連続関数とするとき、以下を満たすCの元 a が存在する。

1. $f(a) = a$
2. $f(b) = b$ なら　$a \subseteq b$

a を f の最小不動点 *(minimum fixed point)* と呼ぶ。

先のデータフローネットワークの表す方程式には、プロセスに対応する関数が含まれている。これらは、連続関数であるので、上の定理より最小の不動点が存在する。すなわち、このデータフローネットワークの実行によってチャネル上に生ずるストリームは，最小不動点として表されるのである。

6.3 表記法の定義

以下では Σ は、高々可算無限個の文字からなる集合で、アルファベットとよぶ。Σから重複を許して有限個または無限個の文字を取り出して1列に並べたものをΣ上の語（word）という。語が含む文字の数を語の長さという。ϵ は空語、すなわち長さ0の語を表す。ϕ は、空集合を表すものとする。特にことわらない限り関数という言葉は、部分関数の意味で用いる。

なお、本章の以下の定義においては、$D, L, L_1, L_2, T, T_1, T_2$は、任意の集合を表すものとしている。

定義 6.3 $\tilde{\Sigma} = \Sigma \cup \{\epsilon\}$ とする。Σ^* はΣ上の有限長の語の全体を表す。ただし、$\epsilon \in \Sigma^*$ である。Σ^ω はΣ上の有限長または無限長の語の全体を表す。ただし $\epsilon \in \Sigma^\omega$ である。

定義 6.4 関数$\nu \in (L \to T), D \subset L$ とし、$\zeta \in (D \to D)$ を恒等写像 $\zeta(x) = x$ とする。このとき、$\nu_{[D]}$は、合成関数 $\nu \cdot \zeta : D \to T$ を表すものとする。

定義 6.5 kを任意の自然数とするとき、$T^{\times k}$ は T の k個の直積を表すものする。すなわち、

$$T^{\times k} = \underbrace{T \times T \times \cdots \times T}_{k}$$

である。

定義 6.6 関数 f_1, f_2は $f_1 : L_1 \to T_1, f_2 : L_2 \to T_2$ かつ $L_1 \cap L_2 = \phi$ とする。このとき、関数 $f_1 \cup f_2$ を以下で定義する。

$$(f_1 \cup f_2)(x) = \begin{cases} f_1(x) & if\ x \in L_1 \\ f_2(x) & if\ x \in L_2 \end{cases}$$

定義 6.7 部分関数 gに対し、$dom(g)$ は g の定義域すなわち以下を表すものとする。

$$dom(g) = \{x \mid g(x) \text{ が定義されている } \}$$

6.4 基本プロセス

基本プロセスは、複数本の入力チャネル、出力チャネルを持つ機械であり、入力チャネルから文字を入力し、出力チャネルに語を出力する。また、基本プロセスでは、入力から出力を作り出す関数は、状態に依存して変化する。

定義 6.8 基本プロセス Pは $P = (I, O, S, q_0, f)$ で表される。ここに、Iは入力チャネルの有限集合、Oは出力チャネルの有限集合で $I \cup O = \phi$。Sは状態の高々可算無限集合。q_0 は初期状態で $q_0 \in S$。fは Pの推移関数といい、以下の型の部分計算可能関数。

$$f : S \times (I \to \tilde{\Sigma}) \longrightarrow S \times (O \to \Sigma^*)$$

ただし、$I = \{i_1, i_2, \cdots, i_m\}, O = \{o_1, o_2, \cdots, o_n\}$ とおき、ζ, ξをそれぞれ、すべての入力チャネル、出力チャネルに空語を対応させる関数すなわち、

$$(\zeta(i_1), \zeta(i_2), \cdots, \zeta(i_m)) = (\epsilon, \epsilon, \cdots, \epsilon),$$

$$(\xi(o_1), \xi(o_2), \cdots, \xi(o_n)) = (\epsilon, \epsilon, \cdots, \epsilon)$$

とするとき、推移関数fは以下（決定性の条件という）を満足しているものとする。

1. 任意の状態 sに対し、$f(s, \zeta) = (s, \xi)$ が成り立つ。すなわち、すべての入力チャネルから空を入力する場合は、出力はすべての出力チャネルで空でなければならない。
2. sを任意の状態とし、$\psi, \hat{\psi} \in (I \to \tilde{\Sigma})$ はともにζではないとする。ある入力チャネル i_kに対し
   $(s, \psi(i_1), \psi(i_2), \cdots, \psi(i_{k-1}), \psi(i_k), \psi(i_{k+1}), \cdots, \psi(i_m)) \in dom(f)$ かつ
   $(s, \hat{\psi}(i_1), \hat{\psi}(i_2), \cdots, \hat{\psi}(i_{k-1}), \hat{\psi}(i_k), \hat{\psi}(i_{k+1}), \cdots, \hat{\psi}(i_m)) \in dom(f)$ かつ
   $\psi(i_k) = \epsilon, \hat{\psi}(i_k) \neq \epsilon$なら、
   適当な $i_j (i_j \in (I \to \tilde{\Sigma})$ かつ $i_j \neq i_k)$ が存在し、 $\psi(i_j), \hat{\psi}(i_j)$ ともに ϵ でなく、かつ $\psi(i_j) \neq \hat{\psi}(i_j)$ 。

この決定性の条件は、あるチャネルから空を入力するか否かは、他のチャネル上のデータとの関係から一意に決定されることを意味している。すなわち、基本プロセスは、入力チャネル上の同一の入力データの組に対し、出力も状態も変化させない空（ζ）入力を除いて、２つ以上の異なる動作をすることはないことを意味している。

入力チャネルの集合Ｉに対しＩの濃度を $|I| = k$ とする。写像の集合 $(I \to \tilde{\Sigma})$ と直積集合$\tilde{\Sigma}^{\times k}$は同型である。本論文では入力、出力のチャネルを $1,2,\cdots,k$ なる整数であると仮定して、基本プロセスの推移関数 f の定義域、値域の任意の要素などを、簡単に $(s, d_1, d_2, \ldots, d_k) \in S \times \tilde{\Sigma}^{\times k}$ などと表すことがある。

例題 6.1 *3*個の入力チャネル $\{1,2,3\}$ と１個の出力チャネル $\{4\}$ をもち、チャネル１の入力文字が、*"G"* か *"L"* かによって、チャネル *2,3* の入力値の大きい値か、小さい値をチャネル *4* に出力する基本プロセス P を作る。これは、$P = (\{1,2,3\}, \{4\}, \{q_0, s_1, s_2\}, q_0, f)$ で（図１）、推移関数 f を以下の通りとすればよい。

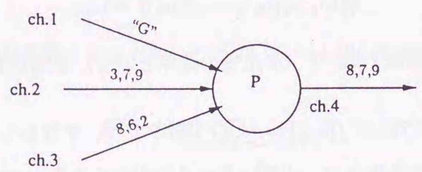


図 2: 基本プロセスの例

$$
\begin{array}{lllll}
\text{状態} & ch.1 & ch.2 & ch.3 & \\
(q_0, & x, & \epsilon, & \epsilon) = & \left\{ \begin{array}{lll} \text{状態} & ch.4 & \\ (s_1, & \epsilon) & if\ x ='' G'' \\ (s_2, & \epsilon) & if\ x ='' L'' \end{array} \right. \\
(s_1, & \epsilon, & y, & z) = & \left\{ \begin{array}{lll} (s_1, & y) & if\ y \geq z \\ (s_1, & z) & else \end{array} \right. \\
(s_2, & \epsilon, & y, & z) = & \left\{ \begin{array}{lll} (s_2, & y) & if\ y \leq z \\ (s_2, & z) & else \end{array} \right.
\end{array}
$$

推移関数 f が決定性の条件を満たしていることは、容易に示される。このプロセスで、例えば入力チャネル *1* に *"G"*, チャネル *2* に *3,7,9* が、チャネル *3* に *8,6,2* がそれぞれ与えられた場合、出力チャネル *4* には、*8,7,9* が得られる。

定義 6.9 基本プロセスの推移関数 f に対し、$f(s, d_1, d_2, \ldots, d_m) = (s', d'_1, d'_2, \ldots, d'_n)$ が成り立つとき、

$$s \xrightarrow[(d'_1, d'_2, \ldots, d'_n)]{(d_1, d_2, \ldots, d_m)} s'$$

と表し、P の推移 *(transition)* という。

定義 6.10 つぎの有限または無限列

$$s_0 \xrightarrow[v_0]{u_0} s_1 \xrightarrow[v_1]{u_1} \cdots \xrightarrow[v_{k-1}]{u_{k-1}} s_k \longrightarrow \cdots$$

は、すべての $i(= 0, 1, 2, \cdots, k, \cdots)$ について、以下がプロセス P の推移であるなら、プロセス P の計算（*computation*）であるという。

$$s_i \xrightarrow[v_i]{u_i} s_{i+1}$$

さらに　$u_0u_1\cdots u_k\cdots$,　$v_0v_1\cdots v_k\cdots$ をそれぞれ P の入力ヒストリ *(history)*, 出力ヒストリという。

以下では、計算におけるヒストリと状態との関係を、

$$s_0 \overset{\substack{u_0u_1\cdots u_k\cdots \\ * \\ v_0v_1\cdots v_k\cdots}}{\longrightarrow} \cdots$$

あるいは

$$s_0 \overset{\substack{u_0u_1\cdots u_k \\ * \\ v_0v_1\cdots v_k}}{\longrightarrow} s_{k+1}$$

と表すことがある。それぞれ、ヒストリが無限長、有限長、の場合を表している。

定義 6.11 $X,Y \in \Sigma^\omega$ とする。X,Yともに無限列のときには、X=Y のときに限り $X \leq Y$ と表す。そうでない場合、 X が Y の接頭部 *(initial segment)* であるとき、すなわち $^\exists Z \in \Sigma^\omega$ に対し Y=XZ になるとき、そのときに限り $X \leq Y$ と表す。

従ってΣ^ωは関係$\leq$の下で半順序集合をなし、ϵ は、Σ^ω の最小の要素である。Σ^ω の要素の任意の増加列 : $X_1 \leq X_2 \leq \ldots \leq X_n \leq \ldots$ は、Σ^ωに最小上界 (least upper bound) すなわち $lim_{i\to\infty}X_i$ を持つので、Σ^ω は関係$\leq$の下で完備な半順序 (c.p.o:complete partial order) 集合をなす[22]。

定義 6.12 A,B は *c.p.o* 集合とする。関数 $f : A \to B$ が単調 *(monotonic)* であるとは、$X \leq Y$ なる任意の $X,Y \in A$ に対し、$f(X) \leq f(Y)$ であることをいう。また、fが連続 *(continuous)* である とは、 任意の増加列 $\{X_i\}_{(i=1,2,\cdots)}$ に対し、$f(lim_{i\to\infty}X_i) = lim_{i\to\infty}f(X_i)$ であることをいう。

定義 6.13 $x,y \in (\Sigma^\omega)^{\times k}$ をそれぞれ、$x = (c_1, c_2, \ldots, c_k)$, $y = (d_1, d_2, \ldots, d_k)$ とする。$c_1 \leq d_1, c_2 \leq d_2, \ldots, c_k \leq d_k$ が成り立つとき $x \leq y$ と表す。このとき、関係 $\leq$ の下で $(\Sigma^\omega)^k$ は *c.p.o* 集合になる。また、$(\epsilon,\epsilon,\cdots,\epsilon) \in (\Sigma^\omega)^k$ を簡単に$\epsilon^{\times k}$ と表す。$\epsilon^{\times k}$は *c.p.o*

集合 $(\Sigma^\omega)^{\times k}$の最小の要素である。ただし、$k$を明示する必要のないときは、$\epsilon^{\times k}$の代わりに$\epsilon$を使うことにする。

定義 6.14 任意の $x = (c_1, c_2, \ldots, c_k)$, $y = (d_1, d_2, \ldots, d_k) \in (\Sigma^*)^{\times k}$ に対し、結合 *(concatenation)* を $xy = (c_1d_1, c_2d_2, \ldots, c_kd_k)$ とする。

基本プロセスが連続関数を表すようにするためには、基本プロセスの計算が途中の入力で推移が進められなくなった場合も含めて、ヒストリの関係として定義しなければならない。そこで以下のように、基本プロセスの計算が表すヒストリの関係をさらに拡張する。

定義 6.15 任意の基本プロセス $P = (I, O, S, q_0, f)$ における関係 *(relation)* $G_1, \hat{G}_1$を以下で定義する。ただし、m,nは基本プロセス Pの入力, 出力チャネル数を表すものとする。

$$\begin{aligned}
G_1 &= \{(u,v) \mid q_0 * \overset{u}{\underset{v}{\longrightarrow}} \cdots \text{または} \\
&\quad {}^\exists s_k \text{に対し } q_0 * \overset{u}{\underset{v}{\longrightarrow}} s_k\} \\
\hat{G}_1 &= \{(uw,v) \mid {}^\exists s_k : q_0 * \overset{u}{\underset{v}{\longrightarrow}} s_k \text{ かつ } w(\neq \epsilon^{\times m}) \in (\Sigma^\omega)^{\times m} \text{で} \\
&\quad {}^\forall u_k(\neq \epsilon^m) \leq w \text{に対し } f(s_k, u_k) \text{ が定義されない}\}
\end{aligned}$$

さらに $(\Sigma^\omega)^{\times(m+n)}$ 上の関係 F_Pを、

$$F_P = G_1 \cup \hat{G}_1$$

で定義する。

G_1は、入力ヒストリをすべて入力し尽くす計算が存在する場合であり、$\hat{G}_1$は、入力ヒストリの途中までは計算があるが、残りの入力に対しては計算が不能である場合に相当する。基本プロセスの推移関数は $S \times (I \to \hat{\Sigma})$ 上の部分関数である。しかし、以下の定理が示すように、ここで定義したヒストリの関係 F_P は全体関数 (total function) になる。

定理 6.2 基本プロセス P における関係 F_P は、$F_P : (\Sigma^\omega)^{\times m} \to (\Sigma^\omega)^{\times n}$ なる型の全体関数である。ただし、m,n は基本プロセス P の入力, 出力チャネル数を表すものとする。

証明）まず、$G_1, \hat{G}_1$ がいずれも $(\Sigma^\omega)^{\times m} \to (\Sigma^\omega)^{\times n}$ なる型の部分関数であることを示す。

(1) $(u,v) \in G_1$ とし、基本プロセス P における有限または無限長の２つの計算 C, C' が、

$$C:\ s_0 \xrightarrow[v_0]{u_0} s_1 \xrightarrow[v_1]{u_1} \cdots \xrightarrow[v_{k-1}]{u_{k-1}} s_k \longrightarrow \cdots$$

$$C':\ s_0 \xrightarrow[x_0]{w_0} s'_1 \xrightarrow[x_1]{w_1} \cdots \xrightarrow[x_{k-1}]{w_{k-1}} s'_k \longrightarrow \cdots$$

であり、u_i, w_i はいずれも $\epsilon^{\times m}$ でなく、$u = u_0u_1\cdots u_k\cdots = w_0w_1\cdots w_k\cdots$ であるとする。このとき $v_0v_1\cdots v_k\cdots = x_0x_1\cdots x_k\cdots$ が成り立つことを示す。

すべての $i(i = 0,1,2,\cdots)$ に対して $u_i = w_i$ の場合は基本プロセスの推移関数の性質から明らかに、$v_0v_1\cdots v_k\cdots = x_0x_1\cdots x_k\cdots$ が成り立つ。そこで適当な k が存在して、すべての $i(0 \le i \le k-1)$ に対し、$s_i = s\prime_i, u_i = w_i, v_i = x_i$ でかつ、$u_k \neq w_k$ であると仮定する。すなわち、$u_k = (\eta_1, \eta_2, \cdots, \eta_m)$, $w_k = (\mu_1, \mu_2, \cdots, \mu_m)$ でかつ適当な入力チャネル l に対し、$\eta_l \neq \mu_l$ であるとする。このとき、以下の２つの場合しかありえない。

1. $\eta_l = \epsilon$ かつ $\mu_l \neq \epsilon$

2. $\eta_l \neq \epsilon$ かつ $\mu_l = \epsilon$

いずれの場合も、基本プロセスにおける推移関数の決定性の条件より、適当な入力チャネル $j \neq l$ が存在し、$\eta_j \neq \epsilon$ かつ $\mu_j \neq \epsilon$ で $\eta_j \neq \mu_j$ が成り立つことになる。これは、

$u_0u_1\cdots u_k\cdots = w_0w_1\cdots w_k\cdots$ の仮定に反する。したがって、計算 $C, C\prime$ は同一の計算であり、G_1 は $(\Sigma^\omega)^{\times m} \to (\Sigma^\omega)^{\times n}$ なる型の部分関数である。

(2) $^\forall(uw, v_1), (uw, v_2) \in \hat{G}_1$ に対して、

$$q_0 * \xrightarrow[v]{u} s_k \text{ かつ } {}^\forall u_k \le w \text{ に対し } f(s_k, u_k) \text{ が未定義}$$

であるとする。このとき、$v_1 = v_2 = v$ でなければならない。したがって、$\hat{G}_1$ は $(\Sigma^\omega)^{\times m} \to (\Sigma^\omega)^{\times n}$ なる型の部分関数である。

$dom(G_1), dom(\hat{G}_1)$ は互いに排反で、$dom(G_1) \cup dom(\hat{G}_1) = (\Sigma^\omega)^{\times m}$ であるので、F_P は全体関数である。

以下ではこの関数 F_P を基本プロセス P のヒストリ関数という。

補題 6.1 基本プロセスのヒストリ関数は、単調関数である。

証明）$X_1 \le X_2$ を $X_1, X_2 \in (I \to \Sigma^\omega)$ なる任意のヒストリとする。X_1, X_2 がともに無限長の場合は $X_1 = X_2$ であり、$F_P(X_1) \le F_P(X_2)$ が成り立つ。そうでない場合、適当なヒストリ U に対して、$X_1U = X_2$ である。$F_P(X_1) = Y_1$ とおけば 、(X_1, Y_1) が関係 $G_1, \tilde{G}_1$ のいずれに属している場合でも、$F_P(X_1U) = F_P(X_2)$ が定義され、$F_P(X_1) \le F_P(X_2)$ が成り立つ。

定理 6.3 基本プロセスのヒストリ関数は、連続関数である。

証明）$(I \to \Sigma^\omega)$ 上の増加列を $\{X_k\}$ すなわち

$$X_0 \le X_1 \le X_2 \cdots \le X_k \le \cdots$$

とする。基本プロセスのこの増加列に対応する計算として、以下の二通りが考えられる。すなわち限りなく計算が続く場合 *(1)* と、途中で（空動作以外に）計算が続かなくなる場合 *(2)* である。

(1)

$$C:\ q_0 * \xrightarrow[Z_0]{X_0} s_1 * \xrightarrow[\mu_1]{\eta_1} \cdots * \xrightarrow[\mu_k]{\eta_k} s_{k+1} \longrightarrow \cdots$$

ここに、$X_1 = X_0\eta_1, X_2 = X_1\eta_2, \cdots, X_k = X_{k-1}\eta_k, \cdots$, かつ、

$Z_1 = Z_0\mu_1, Z_2 = Z_1\mu_2, \cdots, Z_k = Z_{k-1}\mu_k, \cdots$ とする。

このとき $k = 1,2,\cdots$ に対し、$F_P(X_k) = Z_k$ すなわち、$F_P(X_0\eta_1\eta_2\cdots\eta_k) = Z_0\mu_1\mu_2\cdots\mu_k$であるので、

$$\begin{aligned} F_P(\lim_{k\to\infty} X_k) &= F_P(X_0\eta_1\eta_2\cdots\eta_k\cdots) \\ &= Z_0\mu_1\mu_2\cdots\mu_k\cdots \\ &= \lim_{k\to\infty} F_P(X_k) \end{aligned}$$

故に、*(1)* の場合は連続関数の条件を満たす。

(2)

$$C' : q_0 * \xrightarrow[Z_0]{X_0} s_1 * \xrightarrow[\mu_1]{\eta_1} \cdots * \xrightarrow[\mu_k]{\eta_k} s_{k+1}$$

ここに、$X_1 = X_0\eta_1, X_2 = X_1\eta_2, \cdots, X_k = X_{k-1}\eta_k$ 。

さらに、$X_{k+1} = X_k\eta_{k+1}$ かつ $\eta_{k+1} \neq \vec{\epsilon}$で$^{\forall}\theta(\neq \vec{\epsilon}) \leq \eta_{k+1}$ に対し、

$f(s_k, \theta)$ なる推移は定義されないとする。

この場合は、$\lim_{j\to\infty} X_j \in dom\hat{G}_1$であり、以下が成り立つ。

$$\begin{aligned} F_P(\lim_{j\to\infty} X_j) &= F_P(X_0\eta_1\eta_2\cdots\eta_k\eta_{k+1}\cdots) \\ &= F_P(X_0\eta_1\eta_2\cdots\eta_k) \\ &= F_P(X_k) \end{aligned}$$

$^{\forall}l > k$ に対し $F_P(X_l) = F_p(X_k)$ 故に

$$\lim_{j\to\infty} F_P(X_j) = F_P(X_k)$$

したがって、

$$F_P(\lim_{j\to\infty} X_j) = \lim_{j\to\infty} F_P(X_j)$$

が成り立つ。よって、*(2)* の場合も連続関数の条件を満たす。

6.5 決定性データフローネットワーク

前章で導入した基本プロセスのもとに、決定性データフローネットワークを定義し、それが入力出力ヒストリ間のどのような関数を意味するかを考察する。

定義 6.16 決定性データフローネットワーク（以下では簡単にネットワークともいう）N は、$N = (\Pi, \Delta, \Omega, \Theta)$ で表される。ここに、Π は基本プロセスの有限集合 $\Pi = \{P_1, P_2, \cdots, P_k\}$。$\Delta, \Omega, \Theta$はいずれもチャネルの有限集合で、いずれも空ではなく、互いに排反でなければならない。これらをそれぞれ、源泉チャネル集合、吸収チャネル集合、内部チャネル集合という。各基本プロセス P_iを $(I_{P_i}, O_{P_i}, S_{P_i}, q_0^i, f_{P_i})$ とするとき、チャネルは以下の関係を満足しているものとする。

$$\begin{aligned} \bigcup_{P_i\in\Pi} I_{P_i} &= \Delta \cup \Theta \\ I_{P_i} \cap I_{P_j} &= \phi \ (for\ i \neq j) \\ \bigcup_{P_i\in\Pi} O_{P_i} &= \Omega \cup \Theta \\ O_{P_i} \cap O_{P_j} &= \phi \ (for\ i \neq j) \end{aligned}$$

決定性データフローネットワーク N の源泉チャネルは、どの基本プロセスに対してもその出力になっていないチャネルであり、一方吸収チャネルはどの基本プロセスに対してもその入力になっていないチャネルである。内部チャネルは、基本プロセスの入力、出力の両方として現れるチャネルである。

ネットワークにおいて、各基本プロセスは、入力チャネル上のデータに対して推移が可能であれば、それぞれのプロセスが独立に推移を行ない出力チャネルにデータを送出する。チャネルはFIFO の機能を備えたバッファ（無限容量）であり、入力が実行されて

いないデータを記憶しているものとする。この点が、入力側と出力側で action を同時に生起する CCS や CSP と異なるところである。

以下では、ネットワークが意味するヒストリ間の関係を基本プロセスのヒストリ関数に基づいて定義をする。

基本プロセスはヒストリ関数を定義するので、基本プロセス P_i のヒストリ関数を F_i で表す。また、源泉、吸収、内部の各チャネル上のヒストリをそれぞれ i_k, o_k, c_k で表すと、決定性データフローネットワークは、たとえば以下のような連立方程式を表すことになる。

$$F_1(i_1, c_1) = (c_4, o_1) \tag{1}$$

$$F_2(i_2, c_2, c_3) = (o_2, c_1) \tag{2}$$

$$F_3(c_4) = (c_2, c_3) \tag{3}$$

(1) 式は関数 F_1 が、源泉チャネル、内部チャネル上のヒストリ i_1, c_1から、内部チャネル、吸収チャネル上のヒストリ c_4, o_1をつくり出すことを表している。(2),(3) も同様である。

さらに、以下のように、吸収チャネルのヒストリを受けとり、どのチャネルにもヒストリを作り出さない特別な関数 F_Ωを付け加える（図２）。

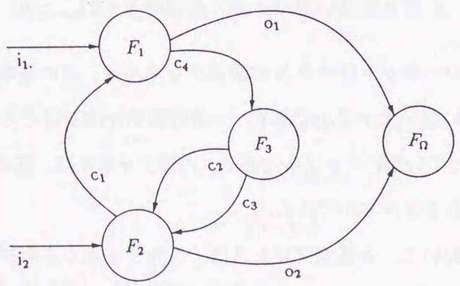


図 3: 吸収チャネルを付加したネットワーク

$$F_\Omega(o_1, o_2) = () \tag{4}$$

これらを以下のように一般化することができる。

各プロセス P_iのヒストリ関数 F_iは、以下の型の関数である。

$$F_i : (I_{P_i} \to \Sigma^\omega) \to (O_{P_i} \to \Sigma^\omega)$$

さらに F_Ωを以下の関数とする。

$$F_\Omega : (\Omega \to \Sigma^\omega) \to \phi$$

ただし、$^\forall v \in (\Omega \to \Sigma^\omega)$ に対して、 $F_\Omega(v) = \phi$ とする。

上記のチャネル上のヒストリの関係は、以下の関数Φとして、統合することができる。

$$\Phi : ((\Delta \cup \Omega \cup \Theta) \to \Sigma^\omega) \to ((\Omega \cup \Theta) \to \Sigma^\omega)$$

ただし、$^\forall U \in ((\Delta \cup \Omega \cup \Theta) \to \Sigma^\omega)$ に対して、

$$\Phi(U) = \bigcup_{P_i \in \Pi} F_i(U_{[I_i]}) \bigcup F_\Omega(U_{[\Omega]})$$

この関数Φは、チャネル $(\Delta \cup \Omega \cup \Theta)$ 上のヒストリに、チャネル $(\Theta \cup \Omega)$ 上のヒストリを対応させる。ただしΩ上のヒストリが何であろうとも、$(\Delta \cup \Theta)$ 上のヒストリから、関数 F_i によって$\Omega \cup \Theta$上ののヒストリをΦの関数値とするものである。

こうして、ネットワーク N は、以下のようなヒストリ間の方程式を表すことになる。

$$\Phi(i_1, i_2, \cdots, i_k, c_1, c_2, \cdots, c_l, o_1, o_2, \cdots, o_m) = (c_1, c_2, \cdots, c_l, o_1, o_2, \cdots, o_m)$$

ただし、i_jは源泉チャネル上のヒストリを、c_jは内部チャネル上のヒストリを、o_jは吸収チャネル上のヒストリを表す。また k,l,m はそれぞれ、チャネル数 $|\Delta|, |\Theta|, |\Omega|$ を表す。

関数Φに対し、源泉チャネルのヒストリを固定して、内部チャネル、出力チャネルのヒストリの関数とみなすことによって、以下の関数$\hat{\Phi}_i$ が得らる。

すなわち、$\iota = (i_1, i_2, \cdots, i_k)$ と置けば、λ記法をまねて

$$\hat{\Phi}_\iota = \lambda c_1 \lambda c_2 \cdots \lambda c_l \lambda o_1 \lambda o_2 \cdots \lambda o_m \Phi(i_1, i_2, \cdots, i_k, c_1, c_2, \cdots, c_l, o_1, o_2, \cdots, o_m)$$

と表現することができる。

関数$\hat{\Phi}_\iota$ は基本プロセスのヒストリ関数を並べたものである。基本プロセスは連続なヒストリ関数を定義するので、以下の補題が成り立つ。

補題 6.2 任意のネットワーク Nにおいて、源泉チャネル上の任意のヒストリ ι に対し、$\hat{\Phi}_\iota$は、$T_N \to T_N$ なる型の連続関数である。ただし、$T_N = (\Omega \cup \Theta) \to \Sigma^\omega$ とする。

証明）略 *(Bird[2]と同様に証明することができる)*

c.p.o集合上の連続関数については最小不動点の存在は良く知られている（例えば [2])。かくして、以下の定理が成り立つことになる。

定理 6.4 任意のネットワーク Nにおいて、源泉チャネル上の任意のヒストリ ι に対し、以下の方程式を満たす最小の解、すなわち$\hat{\Phi}_\iota$の最小の不動点 $(\in (\Theta \cup \Omega) \to \Sigma^\omega)$ が存在する。

$$\hat{\Phi}_\iota(c_1, c_2, \cdots, c_l, o_1, o_2, \cdots, o_m) = (c_1, c_2, \cdots, c_l, o_1, o_2, \cdots, o_m)$$

しかも $\hat{\Phi}_\iota^n(\vec{\epsilon}) = \hat{\Phi}_\iota(\hat{\Phi}_\iota(\cdots(\hat{\Phi}_\iota(\vec{\epsilon})\cdots)))$ $(n$回$)$ とするとき、不動点$(c_1, c_2, \cdots, c_l, o_1, o_2, \cdots, o_m)$は以下で表される。

$$(c_1, c_2, \cdots, c_l, o_1, o_2, \cdots, o_m) = \lim_{n\to\infty} \hat{\Phi}_\iota^n(\vec{\epsilon})$$

証明）略

$\hat{\Phi}_\iota(\vec{\epsilon})$ は、源泉チャネル以外のチャネルはすべて空の状態からネットワークの計算を開始することを意味している。したがってこの定理から、以下のことがいえる。決定性データフローネットワークNにおいて、源泉チャネル上に任意のヒストリ ι が与えられたとき、すべての内部チャネルを空にしてネットワークの実行を開始し、永久に実行し続けるとする。このとき内部チャネル、吸収チャネル上に現れるヒストリは$\hat{\Phi}_\iota$ の最小不動点である。なお、特に不動点が有限長のヒストリの場合については、次の節で考察する。

6.6 ネットワークの実行例

源泉チャネルに与えられた非負の整数kにたいし$k!$の値を計算し吸収チャネルに出力するデータフローネットワークを構成してみる。ここでは、アルファベットΣ は、非負整数の全体であるとする。階乗を乗算の繰り返しで求めるとして、無限状態をもつ基本プロセスを使うこと（プログラムで中間の演算結果を変数に記憶させることに相当）も可能であるが、ここでは、あえてすべてのプロセスを有限状態に限定して設計する。まず基本プロセスとして、Jdg,Prdを以下のように設定する。

- 基本プロセスJdg=$(\{K, KM1, F1\}, \{FC, K0, F0\}, \{q_0, s_1\}, q_0, f_J)$
  ただし、$f_J(S, K, KM1, F1) = (S, FC, K0, F0)$ の関数関係は、表1の推移規則で表されるものとする。

表1 Jdgの推移表
Table 1 A transition table of Jdg.

S	K	KM1	F1	S	FC	K0	F0
q_0	0	ϵ	ϵ	q_0	1	ϵ	ϵ
q_0	$k(\neq 0)$	ϵ	ϵ	s_1	ϵ	k	1
s_1	ϵ	0	$\forall m$	q_0	m	ϵ	ϵ
s_1	ϵ	$k(\neq 0)$	$\forall m$	s_1	ϵ	k	m

表2 Prdの推移表
Table 2 A transition table of Prd.

S	K0	F0	S	KM1	F1
q_0	$\forall k$	$\forall m$	q_0	$k-1$	$m \times k$

この基本プロセスJdgは、最初は状態q_0で入力チャネルKの値を読み込む。２回目以降は状態s_1になり、チャネルKM1,F1から入力する。いずれの状態でも、乗算をさらに繰り返して実行するか否かを判定し、実行するならチャネルK0,F0にデータを送りだし、一方チャネルF１に階乗の値が得られたら、チャネルFCにそ

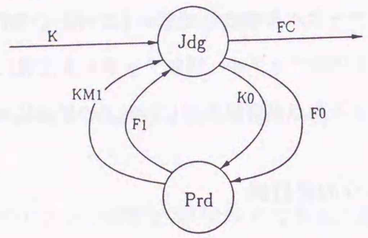


図 4: 階乗計算のネットワーク

の値を送りだすものである。

- 基本プロセス Prd=$(\{K0, F0\}, \{KM1, F1\}, \{q_0\}, q_0, f_P)$

  ただし、$f_P(S, K0, F0) = (S, KM1, F1)$ の関数関係は、表２の推移規則で表されるものとする。

  基本プロセス Prd は、チャネル K0,F0 に値を受け、$F0 \times K0, K0 - 1$ の値をそれぞれチャネル KM1,F1 に送り出すものである。

これら Jdg,Prd はいずれも決定性の推移規則をもち、ヒストリの関数を定義する。

- データフローネットワークの構成

  基本プロセス Jdg,Prd を持つデータフローネットワークを $N = (\{Jdg, Prd\}, \{K\}, \{FC\}, \{KM1, F1, K0, F0\})$ とする（図 3)。基本プロセス Jdg,Prd に対するヒストリ関数をそれぞれ F_J, F_P とし、吸収チャネル FC のヒストリを受ける（なにも関数値を生じない）だけの関数を F_Ω とすれば、以下の関係が得られる。これらの式の右辺は、左辺の式の計算によって作られるデータのチャネル上のヒストリを表している。また、これらの式の左辺で、λによって束縛されていない変数の内容 (チャネルのヒストリ）は、このネットワークの実行時には固定されているものと考える。

表３　ヒストリの変化

Table 3　Transition of the histry.

KM1	F1	K0	F0	FC	KM1	F1	K0	F0	FC
ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	2	1	ϵ
ϵ	ϵ	2	1	ϵ	1	2	2	1	ϵ
1	2	2	1	ϵ	1	2	2·1	1·2	ϵ
1	2	2·1	1·2	ϵ	1·0	2·2	2·1	1·2	ϵ
1·0	2·2	2·1	1·2	ϵ	1·0	2·2	2·1	1·2	2
1·0	2·2	2·1	1·2	2	1·0	2·2	2·1	1·2	2

λ KM1. λ F1. $F_J(K, KM1, F1)$ = (FC,K0,F0)

λ K0 . λ F0. $F_P(K0, F0)$ = (KM1,F1)

λ FC. $F_\Omega(FC)$ = ()

源泉チャネル K に与えられるヒストリをκ とすれば、両辺のチャネル上のヒストリの関係は、以下の式であらわされる。

$$\hat{\Phi}_\kappa(KM1, F1, K0, F0, FC) = (KM1, F1, K0, F0, FC)$$

κを $(K = 2)$ として、プロセスの実行を追いながら各ヒストリを計算してみる (表３)。

たとえば、表 3 の 1 行目は、以下のように解釈できる。源泉チャネル K に２をあたえ、他のすべてのチャネルを空（ϵ）の状態にして各プロセスを実行させると、プロセス Jdg によってチャネル K0,F0 にそれぞれ 2,1 が送り出される。すなわち、チャネル (KM1,F1,K0,F0,FC) が空のヒストリ

$$(\epsilon, \epsilon, \cdots, \epsilon)$$

から

$$(\epsilon, \epsilon, 2, 1, \epsilon)$$

なるヒストリが作られることを表している。これが２行目の左側に示されるヒストリになり、同様の計算が進められ２行目の右側のヒストリが得られる。３行目以降も同様である。

こうして、

$$(KM1, F1, K0, F0, FC)$$
$$= (1\cdot 0,\ 2\cdot 2,\ 2\cdot 1,\ 1\cdot 2,\ 2)\cdots(1)$$

が $\hat{\Phi}_{(2)}$ の不動点であることがわかる。この不動点のうち、FC=2 がこのネットワークから出力されるヒストリである。(1) は $\lim_{n\to\infty}\hat{\Phi}^n_{(2)}(\vec{\epsilon})$ に相当し、$\hat{\Phi}_{(2)}$ の最小不動点である。

6.7 最小不動点の考察

本章では、第４章で得られたヒストリの不動点がデータフローネットワークのどのような実行結果に対応しているのかを調べる。第４章では、ネットワークのヒストリ関数を基本プロセスのヒストリ関数からのみ構成した。以下では、基本プロセスの推移関数や計算を基に、データフローネットワークの推移や計算を定義し、それらとネットワークにおけるヒストリ関数との関係を考える。

定義 6.17 ネットワークを $N = (\Pi, \Delta, \Omega, \Theta)$ とし、Π に属する各プロセス $P_j = (I_{P_j}, O_{P_j}, S_{P_j}, q^j_0, f_{P_j})$ に対し、以下 *(a)* が成り立つとする。

$$\left\{\begin{array}{l}\text{適当な } s^j, s^{j\prime} \in S_{P_j} \text{および}\\ \text{適当な } X_j \in (I_{P_j} \to \hat{\Sigma}), Y_j \in (O_{P_j} \to \Sigma^*)\end{array}\right\} \text{に対し、} f_{P_j}(s^j, X_j) = (s^{j\prime}, Y_j)\cdots(a)$$

このとき、

$$\text{関数}\sigma, \sigma' : \Pi \to \bigcup_{P_i\in\Pi} S_{P_i}$$

を、

$$\sigma(P_i) = s^i, \sigma'(P_i) = s^{i\prime}$$

である関数とし、さらに

$$X = \bigcup_{P_j\in\Pi} X_j\ ,\ Y = \bigcup_{P_j\in\Pi} Y_j$$

とおき、

$$\sigma \overset{X}{\underset{Y}{\Longrightarrow}} \sigma'$$

と表す。これを、ネットワーク N の推移という。

これは、ネットワーク全体の状態がσで、各基本プロセスの入力チャネルから X を入力し、出力チャネル上に Y を出力して、ネットワーク全体の状態がσ'に変化するような各基本プロセスの推移があることを表している。

定義 6.18 つぎの有限または無限列

$$\sigma_0 \overset{U_0}{\underset{V_0}{\Longrightarrow}} \sigma_1 \overset{U_1}{\underset{V_1}{\Longrightarrow}} \cdots \overset{U_{j-1}}{\underset{V_{j-1}}{\Longrightarrow}} \sigma_j \Longrightarrow \cdots$$

は、すべての $i(=0,1,2,\cdots,j,\cdots)$ について、以下がネットワーク N の推移であるなら、ネットワーク N の計算 (*computation*) であるという。

$$\sigma_i \overset{U_i}{\underset{V_i}{\Longrightarrow}} \sigma_{i+1}$$

定義 6.19 ネットワーク N の計算 C (有限または無限列) が以下のようになっているとき、

$$C : \sigma_0 \overset{U_0}{\underset{V_0}{\Longrightarrow}} \sigma_1 \overset{U_1}{\underset{V_1}{\Longrightarrow}} \cdots \overset{U_{j-1}}{\underset{V_{j-1}}{\Longrightarrow}} \sigma_j \Longrightarrow \cdots$$

計算 Cにおける入出力ヒストリと状態との関係を、

$$\sigma_0 \ * \ \overset{U_0U_1\cdots U_j\cdots}{\underset{V_0V_1\cdots V_j\cdots}{\Longrightarrow}} \ \cdots$$

あるいは、

$$\sigma_0 \ * \ \overset{U_0U_1\cdots U_j}{\underset{V_0V_1\cdots V_j}{\Longrightarrow}} \ \sigma_j$$

と表すことがある。

$\hat{\sigma}_0$は、ネットワークのすべての基本プロセス P_jに初期状態を対応させる関数、すなわち$\hat{\sigma}_0(P_j) = q_0^j$とする。このとき以下の計算は、

$$\hat{\sigma}_0 \ * \ \overset{U_0U_1\cdots U_j\cdots}{\underset{V_0V_1\cdots V_j\cdots}{\Longrightarrow}} \ \cdots$$

ネットワークのヒストリ関数$\Phi : ((\Delta \cup \Omega \cup \Theta) \to \Sigma^\omega) \to ((\Omega \cup \Theta) \to \Sigma^\omega)$ に対して、$\Phi((U_0U_1\cdots U_j\cdots) \cup z) = V_0V_1\cdots V_j\cdots$ （ただし $z \in \Omega \to \Sigma^\omega$ は任意のヒストリ）が成り立つことを意味している。計算が有限長の場合についても同様である。

定義 6.20 以下の（有限または無限列の）計算 C_0

$$C_0 : \hat{\sigma}_0 \overset{U_0}{\underset{V_0}{\Longrightarrow}} \sigma_1 \overset{U_1}{\underset{V_1}{\Longrightarrow}} \cdots \overset{U_{j-1}}{\underset{V_{j-1}}{\Longrightarrow}} \sigma_j \quad \Longrightarrow \cdots$$

および源泉チャネル上のヒストリ $\iota \in (\Delta \to \Sigma^\omega)$ に対し、

$U_{0[\Theta]} = \vec{\epsilon}$

$(U_0U_1U_2\cdots U_i)_{[\Delta]} \leq \iota \quad (for\ i = 1, 2, \cdots)$

$$(U_1U_2\cdots U_i)_{[\Theta]} \leq (V_0V_1\cdots V_{i-1})_{[\Theta]} \quad (for\ i = 1, 2, \cdots)$$

を満たすとき、C_0は空チャネルからのιの計算であるという。

すなわち、空チャネルからのιの計算とは、源泉チャネルにヒストリιが与えられ、他のチャネルがすべて空の下で、プロセスをそれぞれの初期状態から実行させたときの計算経過を表している。

定義 6.21 データフローネットワークにおいて、空チャネルからのιの計算を C_0とする。ただしC_0は有限列の計算であるとする。

$$C_0 : \hat{\sigma}_0 \overset{U_0}{\underset{V_0}{\Longrightarrow}} \sigma_1 \overset{U_1}{\underset{V_1}{\Longrightarrow}} \cdots \overset{U_{j-1}}{\underset{V_{j-1}}{\Longrightarrow}} \sigma_j$$

この計算C_0の最後の状態σ_jにおいて、源泉チャネル, 内部チャネルに残されたままで、入力されていない語の組を ξ_{C_0}で表し、C_0の残り列という。すなわち、$\xi_{C_0} \in ((\Delta \cup \Theta) \to \Sigma^\omega)$ で $\iota = (U_0U_1\cdots U_{j-1})_{[\Delta]}\xi_{C_0[\Delta]}$ かつ $(V_0V_1\cdots V_{j-1})_{[\Theta]} = (U_0U_1\cdots U_{j-1})_{[\Theta]}\xi_{C_0[\Theta]}$ である。

以下の定理は、ネットワークのヒストリ関数$\hat{\Phi}_\iota$の不動点の意味を、基本プロセスの計算やチャネルの様相に結び付けて説明している。

定理 6.5 Xは有限長のヒストリで $X \in ((\Theta \cup \Omega) \to \Sigma^*)$ とする。Xが、データフローネットワークの源泉チャネルのヒストリ$\iota \in (\Delta \to \Sigma^\omega)$ における最小不動点、すなわち $\hat{\Phi}_\iota(X) = X$ なる最小の Xであるための必要十分条件は、*(6.5)* を満たす以下の空チャネルからのιの計算 C_0が存在することである。

$$C_0 : \hat{\sigma}_0 \ * \ \overset{U}{\underset{X}{\Longrightarrow}} \ \sigma_j$$

ただし、計算 C_0の残り列をξ_{C_0} とするとき、

(5.1) $\xi_{C_0} \neq \vec{\epsilon} \in ((\Delta \cup \Theta) \to \Sigma^*)$ のとき $h(\neq \vec{\epsilon}) \leq \xi_{C_0}$なるいかなる h に対しても、

$$C' : \sigma_j \ \overset{h}{\underset{w}{\Longrightarrow}} \ \sigma_{j+1}$$

なる計算 C'が存在しない.

証明) $\Rightarrow$) Xは最小不動点であるから、$\hat{\Phi}_\iota^n(\vec{\epsilon}) = \hat{\Phi}_\iota(\hat{\Phi}_\iota(\cdots(\hat{\Phi}_\iota(\vec{\epsilon})\cdots)))$ (n 回) とするとき、

$$X = \lim_{n\to\infty} \hat{\Phi}_\iota^n(\vec{\epsilon})$$

と表せる。しかも、Xは有限長であるから、適当な $j \geq 1$ が存在して、

$$\hat{\Phi}_\iota^{j-1}(\vec{\epsilon}) = \quad X$$
$$\hat{\Phi}_\iota^{j}(\vec{\epsilon}) = \quad \hat{\Phi}_\iota(X) \quad = X$$

一般に $\hat{\Phi}_\iota^{k-1}(\vec{\epsilon}) = W_{k-1}$, $\hat{\Phi}_\iota^{k}(\vec{\epsilon}) = \hat{\Phi}_\iota(W_{k-1}) = W_k$ とおくとき、後者は以下の空チャネルからのιの計算 C_1に相当している。

$$C_1 : \hat{\sigma}_0 \ * \overset{U}{\underset{W_k}{\Longrightarrow}} \sigma_j$$

ただし、$\hat{\Phi}_\iota(W_{k-1}) = W_k$ より、 適当な β が存在して、$U_{[\Theta]}\beta = W_{k-1[\Theta]}$ しかも $\beta \neq \vec{\epsilon}$のとき、$h(\neq \vec{\epsilon}) \leq \beta \cup \xi_{C_1[\Delta]}$なるいかなる h に対しても、

$$C' : \sigma_j \overset{h}{\underset{w}{\Longrightarrow}} \sigma_{j+1}$$

なる計算 C'が存在しない.

したがって、$\hat{\Phi}_\iota^j(\vec{\epsilon}) = \hat{\Phi}_\iota(X) = X$ に対応している空チャネルからのιの計算 C_0 が存在して、

$$C_0 : \hat{\sigma}_0 \ * \overset{U}{\underset{X}{\Longrightarrow}} \sigma_j$$

この場合、$X = W_k = W_{k-1}$に相当するので、$\beta = \xi_{C_0[\Theta]}$ である。すなわち $X_{[\Theta]} = U_{[\Theta]}\xi_{C_0[\Theta]}$ で、$\xi_{C_0} \neq \vec{\epsilon}$のとき $h \leq \xi_C$ なるいかなる $h(\neq \vec{\epsilon})$ に対しても、

$$C' : \sigma_j \overset{h}{\underset{w}{\Longrightarrow}} \sigma_{j+1}$$

なる計算 C'が存在しない。 したがって、計算 C_0は条件 *(6.5)* を満足する。

$\Leftarrow$) 条件 *(6.5)* を満足する空チャネルからのιの計算 C_0が存在し、計算 C_0の残り列を ξ_{C_0} (すなわち $U_{[\Theta]}\xi_{C_0[\Theta]} = X_{[\Theta]}$)であるとする。したがって、$\Phi(U \cup X_{[\Omega]}) = X$ が成り立つ。ξ_{C_0}が空でも、あるいは空でなく条件 *(6.5)* を満たす場合でも、$\hat{\Phi}_\iota(U_{[\Theta]}\xi_{C_0[\Theta]} \cup X_{[\Omega]}) = X$ したがって $\hat{\Phi}_\iota(X_{[\Theta]} \cup X_{[\Omega]}) = \hat{\Phi}_\iota(X) = X$ である。C_0が空チャネルからのιの計算であることは、適当な kが存在して、$\hat{\Phi}_\iota^k(\vec{\epsilon}) = X$ であることを意味している。すなわち、Xは最小の不動点である。

この定理から、データフローネットワークのヒストリ関数$\hat{\Phi}_\iota$の最小不動点の意味を、以下のように述べることができる。

データフローネットワークが、内部チャネルが空の状態で計算を開始するものとする。このとき、源泉チャネルに与えられた入力に対し、永久にチャネル上に出力を出し続けるなら、その無限長の出力が最小不動点である。一方すべての内部チャネルが空 ($\xi_{C_0[\Theta]} = \vec{\epsilon}$) になることがあるなら、その時点までの出力ヒストリが最小不動点であり、もし、このデータフローネットワークがすべてのプロセスで入力不能状態になったら、やはりその時点までの出力ヒストリが最小不動点である。

6.8 不動点の解釈についてのまとめ

決定性データフローネットワークを形式化し、ヒストリとしての最小不動点の存在を示した。筆者のモデルでは、ネットワークの計算するヒストリ関数は連続関数になること、すなわち、任意の決定性データフローネットワークが最小不動点をもつことが判った。さらに、この不動点に対し操作的意味を対応させることができた。すなわちネットワークのすべての内部チャネルが空になるか、あるいはチャネル上のデータをどの基本プロセスも入力出来ない状態に達するなら、有限長の不動点が存在するということができる。

本論文の決定性データフローネットワークがKahnのプログラミング言語[22]やStella[41]

の基本機能を包含していることを示すことができるが、本論文ではほとんど触れなかった。しかし一般にプログラミング言語が、入力チャネルが空か否か（すなわちチャネルにデータが到着しているか否か）を判定する機能を含んでいれば、入力, 出力ヒストリの関係は、チャネル上にデータが発生するタイミングに依存する。すなわち、同一入力のヒストリに対し、複数の出力ヒストリが対応し、その関係は本論文における基本プロセスのような部分関数にはならない。そのような並列処理言語と同等な機能をもつ決定性(ヒストリとしては非決定性）のネットワークのもとでの意味論が課題である。

7 IO正則表現によるデータフローネットワークの検証

7.1 データフローネットワークの性質

データフローネットワークは，構造化分析 [6] のデータフローダイアグラムに現れるように，仕様の表現としても有力である．データフローで表現された仕様をプログラム化することをねらいとした言語は，すでにいくつか開発されている [23][41][49]．また，データフローネットワークでモデル化することができる問題領域の一つとして，共有メモリを持たない環境での分散アルゴリズム [40] などがある．このようなことから，データフローネットワークのもとでの正当性検証が必要とされる．筆者は，プロセスやストリームの表現法を含んだ実際的な検証方法を開発した．

現在までにデータフローネットワークの正当性の検証に関して，複数の提案がなされてきている．同期の並行プロセスに対する検証に比べて，データフローネットワークのような非同期プロセスに対するものは多くはない．代表的のものとして，Kahn[22] とNguyen[32] が挙げられよう．Kahn は，決定性データフローネットワークに対し，各プロセスの入力ストリームと出力ストリームの関係を方程式で表現し，ストリームが接頭語の関係のもとで半順序関係を保つことを利用して，連立方程式の最小不動点がネットワークの表すストリームであることを示した．Kahn は更に，不動点帰納法を用いて証明する簡単な例をあげている．しかし，彼の証明にはストリームやプロセスの仕様を統一的に表現する手段を持たないので，ネットワーク全体を一括して扱うことになり，現実的な問題に適用するのはやや困難であろうと思われる．Nguyen らの方法 [32] は，時制論理（Temporal　Logic）に基づいてプロセスの振る舞を表現し検証するものである．しかし彼らの方法では，プロセス内の状態推移と，チャネルを通してのプロセス間の関係が表現として分離していないので，検証手続きを手順化することは困難である．また，時制論理では静的な論理と動的な論理が混在しており，データのもつ前後関係はかえって見えにくくなっている．

我々の方法は，ストリームおよびプロセスの振る舞を有限状態機械の正則表現 [42] と同様な表現（ＩＯ正則式）で表し，ネットワーク全体の検証は，個々のチャネルに対し，

出力ストリームと入力ストリームが一致することを証明しつつ進める方法である．この方法は，データフローネットワークの解であるストリームが，不動点であるという考えに基づいたものである．正則表現によることから，ストリームを，再帰なしの直感的に把握しやすい表現で表すことができ，また，データの生起する（動的な）順序関係とデータ間の（静的な）論理関係を区別して表現することができる．なお，IO 正則式は特に制限を設けなければ，非決定性の振る舞を含むことになる点に注意しなければならない．我々は，式が決定性を満たすための条件を求め，その下で検証の方法論を展開している．

検証の手順は，以下のように要約できる．まず，各プロセスの入力ストリームを仮定し，それらの入力ストリームからプロセスの出力ストリームの表現を作り出す．続いて各チャネルについて出力ストリームと入力ストリームが同一であることを証明する．更に，すべてのチャネル上のストリームについて，それが最小の不動点なることを示し，最後にこれらのストリームが仕様を満足することを証明する．このように，我々の方法は，すべてのストリームを求めた上で目的とする仕様の証明を行う方式である．

本節では個々のプロセス内の証明には触れず，ネットワーク全体の検証のみに的をしぼって報告する．

以下，第２項では，IO 正則式の表現を述べ，第３項では，そのIO 正則式の意味を集合に基づいて定義し，同時に，IO 正則式が満たすべき決定性の条件を定義する．第４項では，特に，プロセスへのストリームの代入に対する意味を述べ，第５項では，正当性検証の手順，第６項では検証の具体例として，フィードバックを含むプロセス間通信の例，分散アルゴリズムの例（最大値発見問題）をあげている．

7.2 プロセスに対するＩＯ正則式

入力，出力のストリーム関係を表す式としてＩＯ正則式 (IO-Regular Expression) を定義し，その具体的な表現法について述べる．

一般にプロセスは，ストリームを入力する入力チャネル，ストリームを出力する出力チャネルをそれぞれいくつか持つ．ただしいずれのプロセスも，入力チャネルと出力チャネルが同一のチャネルであることは許されないものとする．特別なチャネルとして，ネットワーク外からの入力を受信する外部入力チャネル，ネットワーク外への出力を送信する外部出力チャネルを持つことがある．

プロセスは局所チャネルという特別のチャネルを持つことができる．これは，プロセス内のデータ記憶に使うことが可能なチャネルで，出力チャネルと同様の扱いになるが，他のプロセスへの入力チャネルにはならないものである．以下では，出力チャネル，出力ストリームという言葉は，特に断りがない限り，この局所チャネル，局所チャネル上のストリームを含んだ意味で用いる．

以下で定義される基本項を基に，選択，連接，反復などによって表現した項を IO 正則式という．プロセスの振る舞を表すＩＯ正則式をプロセス式ということがある．IO 正則式はチャネル上のストリームを表現するときにも用い，その場合はストリーム式ともいう．

定義 7.1

< ２項演算子 >::= +| − | × | ÷ |*mod*
< 算術因子 >::=< 定数 > | < 変数 > | < 関数式 >
| (< 算術式 >)
< 算術式 >::=< 算術因子 >
| < 算術因子 >< ２項演算子 >< 算術因子 >

定数は整数と文字列，変数は後述するチャネル名などの名前，関数式は *abs(x),sqrt(x)* など計算可能な関数の呼び出しとする．

< 比較演算子 >::=≤ | ≥ | ≠ | > | < | =
< 比較式 >::=< 算術式 >< 比較演算子 >< 算術式 >
< 論理因子 >::=< 比較式 > | (< 論理式 >)
< 論理式 >::=< 論理因子 > |¬ < 論理因子 >
| < 論理因子 > ∧ < 論理因子 >
| < 論理因子 > ∨ < 論理因子 >
| < 論理因子 >⇒< 論理因子 >
< 入出力関係式 >::=< 論理式 >
< 基本項 >:: = [< 入出力関係式 >]

なお，上記定義に表れる記号はいずれも，通常の数学的な意味を表すものとする．

< 入出力関係式 >は，入力チャネルから入力するデータと出力チャネルに出力するデー

タの関係を表す論理式であり，１度の入力出力の動作によって引き起こされるデータ（以降，入力，出力の単位となるデータをアイテムと呼ぶ）の関係を表す．入力, 出力されるアイテムを，このプロセスの入力チャネル名，出力チャネル名によって表すこととし，これらの表記を含む論理式が< 入出力関係式 >である．

< 入出力関係式 >内の< 変数 >は，ストリーム内のデータを表現するためのものであり，以下の表記から成る．

- $\alpha?$:αは入力チャネル名．$\alpha?$を現入力といい，プロセスがチャネルαから次に入力するデータを意味する．そしてこのデータは，入力としてチャネルから取り出され，入力済みストリームに加えられる．
- β :β は出力チャネル名．現出力といい，現時点でこのチャネルに出力し，出力ストリームに加えられるデータを表す．
- $\gamma\natural$:補助変数と呼ばれ，γ は任意の名前である．ストリーム内にデータを一時的に記憶しておく変数である．この補助変数は，出力チャネルと同じように扱う（出力チャネルと同様に，この変数へのデータの割り当てを出力ともいう）が，出力チャネルと違い，IO 正則式の意味する集合（後述：定義 3.8）には直接含まれない．しかも，これが表れるストリーム内だけで有効であり，異なるストリームでは異なる変数として扱われる．すなわち，ストリーム上でデータの橋渡しをするための補助に過ぎない．
- $\overline{\alpha?}, \overline{\overline{\alpha?}}, \overline{\beta}, \overline{\overline{\beta}}, \overline{\gamma\natural}, \overline{\overline{\gamma\natural}}$:これらは，入力，出力チャネル（補助変数）それぞれに対し，この時点で，すでに入力, 出力済みストリームの先頭（直前に入力，出力されたもの），２番目などのデータを意味するものとする．これらをそれぞれ，第１既入力，第２既入力，第１既出力，第２既出力という．第３，第４ なども同様の表現とする．これらの表記に対しては，新たな入出力の動作は引き起こされない．

項は基本項を基に，選択，連接，反復の３通りの演算によって項から構成される．

定義 7.2

$$\begin{aligned} &< \text{項} > ::= \ < \text{基本項} > \mid [< \text{項} >] \\ &\quad \mid < \text{項} > \vee < \text{項} > \mid < \text{項} >< \text{項} > \\ &\quad \mid < \text{項} >^{*} \end{aligned}$$

IO 正則式は，上で定義された項をもとに，以下で表される．

定義 7.3

$$\begin{aligned} &< IO \text{ 正則式} > \ ::= < \text{項} > \\ &\quad \mid < \text{項} >^{\omega} \mid < \text{項} >< \text{項} >^{\omega} \\ &\quad \mid < IO \text{ 正則式} > \vee < IO \text{ 正則式} > \end{aligned}$$

選択 ($\vee$), 連接 ($\cdot$), 反復 ($*$) は，記号列の正則式 (regular expression)[42] におけるそれらと同様の意味である．なお，演算の優先度は反復 (又はω)，連接，選択の順に高いものとし，同一演算については左から順に評価されるものとする．必要とあらば，カッコ[ ]によって優先演算を表現することができる．ωは，連接の限りない繰り返しを意味する．ただし，これらについて，プロセス式とストリーム式での違いもあり，更に形式的な意味は後述する．

例題 7.1 例えば，以下の式は，入力チャネル in，出力チャネル　out を持つプロセスの IO 正則式の例である．これは，入力チャネル in からの正のデータはすべて出力チャネル out にそのまま流し，入力が 0 のとき繰り返しを終えることを表している．

$$[in? > 0 \wedge out = in?]^{*}[in? = 0]$$

定義 7.4 既入力，既出力を表す変数に割り当てられるアイテムが，すべてこの IO 正則式内の変数として出現するとき，この IO 正則式は独立であるという．

たとえば，　$[\overline{out} > 0][out = \overline{out} + 1]$　は，既出力$\overline{out}$に割り当てられるアイテムが，この式の前に与えられなければならない．したがって，この IO 正則式は独立ではない．

定義 7.5 *IO*正則式 f に対し，*f*に現れる入力チャネル $In = \{in_1?, in_2?, \ldots, in_k?\}$，出力チャネル *(*入力チャネル以外のチャネルで*f*に表れるもの*)*$Out = \{out_1, out_2, \ldots, out_m\}$ を明記するとき，以下のように記述する．

$$f(In; Out) \quad \text{または，}$$

$$f(in_1?, in_2?, \ldots, in_k?; out_1, out_2, \ldots, out_m)$$

7.3 IO 正則式の意味

7.3.1 IO 正則式の表す n 項関係

任意のアイテムの (有限又は無限) 集合Δに対し，$\hat{\Delta}$を以下で定義する．

定義 7.6

$$\hat{\Delta} = \Delta \cup \{\vdash\}$$

ここに，$\vdash$ はストリームにおける過去と現時点との境界を表す記号で，Δには属さない特殊な記号である．

IO 正則式の項は，入力済み，出力済みのデータや次に入力可能データをもそのストリームに含むので，一般に，各チャネルのストリームは以下で定義されるΔ′の部分集合を表すことになる．

定義 7.7

$$\Delta' = \Delta^* \vdash \Delta^\infty$$

但し，

$$\Delta^\infty = \Delta^* \cup \Delta^\omega$$

$$\Delta^\omega = \lim_{k \to \infty} \Delta^k$$

とする．

本章では，簡単化するため，ネットワーク内のチャネルの名前が自然数 $(1, 2, \ldots, n)$ によって付けられているものとして扱う．

定義 7.8

$$\vec{\Delta'} = \Delta'_1 \times \Delta'_2 \times \ldots \times \Delta'_n$$

但し，Δ_iは各チャネル i 上のアイテム集合，n はネットワーク全体のチャネル数を表す．以下では$\vec{\Delta'}$ の要素をストリームベクトルと呼ぶ．

定義 7.9 Xを任意のストリームベクトルとするとき，$X_{(i)}$は X のチャネル iの要素を表す．チャネルの集合 $Ch = \{c_1, c_2, \ldots, c_k\}$ に対し，$X = (v_1, \ldots, v_{c_1}, \ldots, v_{c_2}, \ldots, v_{c_k}, \ldots, v_n)$ のとき，

$$X_{(Ch)} = (v_{c_1}, v_{c_2}, \ldots, v_{c_k})$$

とする．また，V を任意のストリームベクトル集合とするとき，V のチャネル iへの射影 $V_{(i)}$は以下で定義される．

$$V_{(i)} = \{v_i | (v_1, v_2, \ldots, v_i, \ldots, v_n) \in V\}$$

チャネル集合 Ch に対する射影 $V_{(Ch)}$も要素がストリームベクトルになることを除いて同様とする．

空のアイテム列 ϵ，空ストリームの集合Δ^H，空ストリームベクトル集合 $\vec{\epsilon}$を以下のように表す．

定義 7.10 (空アイテム列) ϵ は長さ 0 のアイテム列，すなわち空アイテム列を表すものとする．

定義 7.11 (空ストリーム集合)

$$\Delta^H = \Delta^* \vdash$$

定義 7.12 (空ストリームベクトル集合)

$$\vec{\epsilon} = \Delta_1^H \times \Delta_2^H \times \ldots \times \Delta_n^H$$

定義 7.13 $x,y \in \Delta'$ に対し，$\exists w \in \Delta^{\infty} : y = xw$ が成り立つとき，x は y の接頭部であるといい，

$$x \sqsubseteq y \quad \text{と表す.}$$

(さらに　$x \neq y$ なら　$x \sqsubset y$　と表す)

$U,V \in \vec{\Delta'}$ に対し，

$$U_{(i)} \sqsubseteq V_{(i)} (1 \leq i \leq n)$$

が成り立つとき，

$$U \sqsubseteq V \quad \text{と表す.}$$

$A,B \subset \vec{\Delta'}$ に対し，

$$\forall U \in A : \exists V \in B : U \sqsubseteq V \quad \text{かつ}$$

$$\forall V \in B : \exists U \in A : U \sqsubseteq V$$

が成り立つとき，

$$A \sqsubseteq B \quad \text{と表す.}$$

入出力関係式において，現入力 $(\alpha?)$, 現出力 (β) によって表されるデータは$\vdash$の右側のアイテムになる．既入力, 既出力 $(\overline{\alpha}, \overline{\overline{\alpha}}$など) は，$\vdash$の左側のアイテムになる．

入出力関係式の意味する集合を表すために，以下の記述法を用いる．

入出力関係式 h を $h(d_1;d_2;\ldots;d_n)$ と表す．但し，$d_i (i = 1,2,\ldots,n)$ は，i 番目のチャネルに対する複数の表記の並びを意味するものとする．すなわち，

$$d_i =< \overline{\delta_i}^{(j_i)}, \ldots, \overline{\overline{\delta_i}}, \overline{\delta_i}, D_i >$$

である．j_iは，その式に現れる第 k 既入力（出力）の値 k の最大値である．また，各 $\overline{\delta_i}^{(k)}$ は，第 k 既入力（出力）が式に現れないときは，空である．D_iは，空又は，$\delta_i?$,δ_i のいずれかである．

例えば，ネットワークが 3 つのチャネルを持ち，それらの名前が ina,inb,outa で，この順に順序付けられているとする．ある入出力関係式 h が，

$$ina? > 0 \wedge ina? = outa + \overline{ina}$$

のとき，これを

$$h(< \overline{ina}, ina? >; <>; < outa >)$$

と表す．更に，例えば，$h(< 3,4 >; <>; < outa >)$ とすれば，入出力関係式に現れる $\overline{ina}, ina?$をそれぞれ 3,4 に置き換えた式

$$h(< 3,4 >; <>; < outa >) \equiv 4 > 0 \wedge 4 = outa + 3$$

を意味することとする．

$$\text{基本項 } t = [h(d_1;d_2;\ldots;d_n)] \tag{1}$$

$$\text{ただし } d_i =< \overline{\delta_i}^{(j_i)}, \ldots, \overline{\overline{\delta_i}}, \overline{\delta_i}, D_i >$$

$$(1 \leq i \leq n)$$

に対し，t の表す n 項関係すなわち t の意味 $\tau(t) \subset \vec{\Delta'}$ を，次のように定義する．

定義 7.14

$$\tau([h(d_1;d_2;\ldots;d_n)])$$

はすべての $i (1 \leq i \leq n)$ について，以下のいずれかが成り立つストリームベクトル X の集合である．

- $D_i = \delta?$又は$D_i = \delta$のとき,

$$X_{(i)} = u\Gamma_i^{(j_i)} \ldots \Gamma_i{}^2\Gamma_i{}^1 \vdash \Gamma_i{}^0$$

- D_i=空 のとき,

$$X_{(i)} = u\Gamma_i^{(j_i)} \ldots \Gamma_i{}^2\Gamma_i{}^1 \vdash$$

但し $\Gamma_i^k \in \Delta_i$ $(0 \leq k \leq j_i)$, $u \in \Delta_i^*$ とし，更に$\gamma_i =< \Gamma_i{}^{(j_i)}, \ldots, \Gamma_i{}^2, \Gamma_i{}^1, \Gamma_i{}^0 >$ とおくとき， 式 *(1)* の d_iをγ_iで置き換えた式$h(\gamma_1;\gamma_2;\ldots;\gamma_n)$が,

$$h(\gamma_1;\gamma_2;\ldots;\gamma_n) = true$$

を満たす.

すなわち，$\tau(t)$ は，現時点までに（この基本項の実現結果も含めて）チャネル上に生じたストリームのベクトル集合を表すことになる．$\vdash$の右側の列は，この基本項の実現によって生じたアイテムで，$\vdash$の左側は，この基本項より以前に生み出された可能性のあるアイテム列になる．

$c \in \Delta^* \vdash \Delta^*, d \in \Delta^* \vdash \Delta^\infty$ とするとき，連接 $c \cdot d$ を以下で定義する．

定義 7.15

$$c \cdot d = \begin{cases} w \vdash xy \\ \quad (if \quad c = w \vdash x \wedge d = wx \vdash y) \\ \text{未定義} \\ \quad (otherwise) \end{cases}$$

以下で，L, M は任意の項とし，R, S は任意の IO 正則式とする．連接 LR に対し，その意味を以下で定義する．

定義 7.16

$$\begin{aligned} \hat{\tau}(LR) = \quad & \{(l_1, l_2, \ldots, l_n) | \\ & l_i = x_{(i)} \cdot y_{(i)} \in \Delta_i'(1 \leq i \leq n) \\ & \wedge x \in \tau(L) \wedge y \in \tau(R)\} \end{aligned}$$

によって定義される$\hat{\tau}$を用いて,

$$\begin{aligned} \text{ストリーム式} \quad & \text{では,} \\ \tau(LR) = \quad & \hat{\tau}(LR) \\ \text{プロセス式} \quad & \text{では,} \\ \tau(LR) = \quad & \begin{cases} \hat{\tau}(LR) \\ \quad (\text{項 } R \text{ が非入力駆動のとき}) \\ \tau(L) \cup \hat{\tau}(LR) \\ \quad (\text{そうでないとき}) \end{cases} \end{aligned}$$

と定義する．ただし，項 R が非入力駆動であるとは,

$$\begin{aligned} \forall U \quad & \in C(R)_{(In)} : \forall \text{ 入力チャネル } i \in In : \\ & U_{(i)} \in \Delta^* \vdash \end{aligned}$$

を満たすことをいう（ただし In は入力チャネル集合,$C(R)$ は後述する実行条件集合である）．すなわち，項 R の出力が入力チャネルの現入力に依存しないで決定されることを意味する．

なお、$\tau(LMS), \tau((LM)S)$ はいずれも、$\tau(L(MS))$ を意味するものとする。

この定義に示されるように，プロセス式は全域関数でもある必要があるので，連接でつなげられた式の接頭部もその意味集合に含め，入力が途中で打ち切られたのと同じ状況をも包含するようにしている．しかし，出力が現入力に依存しない基本項（自立的にプロセスが出力する基本項）は, 打ち切りなしに次の基本項へ自動的に継続するものとして扱う．

選択に対し, その意味を以下で定義する．

定義 7.17

$$\tau(R \vee S) = \tau(R) \cup \tau(S)$$

定義 7.18 有限または無限の列

$$U_1 \sqsubseteq U_2 \sqsubseteq \ldots \sqsubseteq U_i \sqsubseteq \ldots \qquad \text{に対し,}$$

$$\forall i : U_i \sqsubseteq Z \quad \text{かつ}$$

$$\forall i : U_i \sqsubseteq W \Longrightarrow Z \sqsubseteq W$$

のとき，Zを$U_1, U_2, \ldots, U_i, \ldots$の上限という．

無限列の場合は，上限を

$$\lim_{k \to \infty} U_k$$

と表す．

反復($*$およびω)に対し，その意味を以下で定義する．

定義 7.19

$$\tau(L^*) = \begin{cases} \bigcup_{0 \leq k < \infty} \tau(L^k) \\ \quad \text{(ストリーム式のとき)} \\ \lim_{k \to \infty} \tau(L^k) \\ \quad \text{(プロセス式のとき)} \end{cases}$$

$$\tau(L^\omega) \quad = \lim_{k \to \infty} \tau(L^k)$$
(ストリーム式，プロセス式のいずれも)

但し，$k = 1, 2, \ldots$に対し $L^k = L^{(k-1)}L$
$k = 0$に対し $\tau(L^k) = \tilde{\epsilon}$とする

この定義に示されるように，我々は，ストリーム式ではL^*,L^ωをそれぞれ有限回の反復，無限回の反復として，異なるものとして扱う[42]．すなわち，前者は有限回で終了するもの，後者は永久に繰り返されるものであり，個々のストリームでは，あらかじめ(有限回で終了か，永久反復かの)区別がなされているものとして扱うことにする．

なお，ストリームとプロセスで，L^ωに対する定義式が同一ではあるが，連接の意味が異なるので，それに伴って意味が異なる．すなわち，プロセス式では，ストリームベクトルの接頭部をも含むのに対し，ストリーム式では，無限列そのものだけを意味する．また，プロセス式では，$*$とωは同一の意味を持つことになる．

定義 7.20 2つのIO正則式t, sに対し，

$$\tau(t) = \tau(s) \quad \text{ならばそのときに限り}$$

$$t \cong s \quad \text{と表す．}$$

7.4 実行条件集合と決定性

本節では，IO正則式の決定性について考える．

定義 7.21 独立なIO正則式tの含む入力チャネル集合をIn,出力チャネル集合をOutとする．

$$\forall U, V \in \tau(t) : \begin{array}{l} U_{(In)} \sqsubset V_{(In)} \\ \quad \text{なら} \quad U_{(Out)} \sqsubseteq V_{(Out)} \\ \quad (U_{(In)} = V_{(In)} \text{のときは,} \\ \quad U_{(Out)} \sqsubseteq V_{(Out)} \quad \text{または} \\ \quad V_{(Out)} \sqsubseteq U_{(Out)}) \end{array}$$

が成り立つとき，IO正則式tは単調であるという．

この単調性は，過去の出力済みストリームベクトルが現時点までの出力ストリームベクトルの接頭部であるという，IO正則式がプログラムのモデルであるために当然満たすべき条件である．ただしこの性質は，同一の入力ストリームベクトルから，複数の出力ストリームベクトルが得られる場合も包含している．それは，IO正則式の意味する集合には，出力が途中で終了した場合（即ち接頭部）も含んでいることがあるからである．そこで，プロセス式が（ストリームベクトルからストリームベクトルへの）関数，すなわち，同一の入力ストリームからは，同一の出力ストリームが得られる，という性質を持つためには，以下の条件を満足しなければならない．

定義 7.22 独立なIO正則式tの含む入力チャネル集合をIn,出力チャネル集合をOutとする．tが単調であり，かつ，以下の条件を満たすとき，tは関数的であるという．

$$\forall U, V \in \tau(t) : \begin{array}{l} U_{(In)} = V_{(In)} \\ \quad \Rightarrow \quad U_{(Out)} = V_{(Out)} \end{array}$$

単調や関数的という性質は，IO 正則式の意味集合からいえる性質である．これらの性質が IO 正則式の表現にどのように表れるかを，以下で考察する．

定義 7.23

$$x = u \vdash v \quad (u \in \Delta^*, v \in \Delta^\infty)$$

のとき

$$\underline{x} = u \vdash a_0$$

$$\text{但し} \quad a_0 = \begin{cases} \epsilon & (if\ v = \epsilon) \\ a & (if\ v = aw, a \in \Delta) \end{cases}$$

また，

$$\tilde{x} = u \vdash$$

と定義する．

項 t に対し，以下のように,t の実行条件集合 $C(t)$ を定義する．これは，項が実行されるために，既入力, 既出力，現入力アイテムの満足すべき条件を意味している．

定義 7.24

$$C(t) = \{(\chi_1, \chi_2, \ldots, \chi_n) | X \in \tau(t) \wedge$$

チャネル i(i 番目のチャネル) が

入力チャネルなら $\chi_i = \underline{X}_{(i)}$,

出力チャネルなら $\chi_i = \tilde{X}_{(i)}\}$

例えば，r, s をそれぞれ入力チャネル，出力チャネルとするとき，

$$t = [\quad r? > 0 \wedge s = r?]^* [r? = 0 \wedge s = r?]$$

の実行条件集合は，

$$C(t) = \{(u \vdash h, v \vdash) | h \geq 0 \wedge u \in \Delta_r^* \wedge v \in \Delta_s^*\}$$

となる（但し，対の 1 番目が入力チャネル r, 2 番目が出力チャネル s に対応させている）．この時点の入力チャネル r の現入力が，この C(t) を満足しないときは，項 t の実行はされず，その地点で他に実行できる項がなければ停止状態になる．

一般に IO 正則式においては，現出力が非決定的に得られる可能性を含んでいる．たとえば，プロセスの同一状態，同一入力から，複数の異なる出力アイテムが得られる可能性も存在する．そこで，本稿の IO 正則式では，そのような性質を持たないものを対象とする．そのために IO 正則式が満たさなければならない条件を，以下で考える．

定義 7.25 2 つのアイテム列 u,v が互いに他の接頭部になっていないとき，u,v は分離しているといい

$$u \bowtie v$$

と表す．更に，以下が成り立つとき，

$\forall A \in C(t_1): \quad \forall B \in C(t_2): \quad \exists i (1 \leq i \leq n):$
$A_{(i)} \bowtie B_{(i)}$

2 つの項 t_1, t_2 が分離しているといい，

$$t_1 \bowtie t_2$$

と表す．

決定性動作を保証するために，IO 正則式における基本項，選択，反復，連接は，以下のような条件を満たすものとする．

1. 基本項 t について：
$\forall X, Y \in \tau(t)$: $\exists$ チャネル i : $\exists u \in \Delta_i^*$:
$(X_{(i)} = u \vdash \Gamma_i \wedge Y_{(i)} = u \vdash \Psi_i \wedge \Gamma_i \neq \Psi_i)$
ならば
($\exists$ 入力チャネル k : $\underline{X}_{(k)} \bowtie \underline{Y}_{(k)}$
又は
$\exists$ 出力チャネル $k(\neq i)$: $\tilde{X}_{(k)} \bowtie \tilde{X}_{(k)}$)
すなわち，この項での現出力アイテムが異なるなら，既出力，現入力，既入力のなかに，異なるものが存在しなければならない．

2. 選択について：

$t_1 \vee t_2$ に対し，$t_1 \bowtie t_2$ すなわち，t_1,t_2　　が分離していること．

3. 反復 (およびω) について:

(t^*又はt^ω) なるtが，以下の (a),(b) を満たす．

(a) $\tau(t) \cap \vec{\epsilon} = \phi$　(空)

(b) $t = s_1 \vee s_2 \vee \ldots \vee s_m$で，$s_k = t_1 t_2^*$

なるs_kに対しては，$t \bowtie t_2$すなわち，

t,t_2 が分離していること．

4. 連接について：

$(s_1 \vee s_2 \vee \ldots \vee s_m)t$ なる連接で、$s_k = t_1 t_2^*$ なるs_kに対しては，$t_2 \bowtie t$ すなわち，t_2 ,t　が分離していること．

(1)-(4) を決定性条件という．プロセスの動作が各時点で一意的に決定されないデータフローネットワークが，構成的 (compositional) ではない [24][25] という望ましくない (すなわち，プロセス式が単調でない) 性質を持つことがあることは既に知られている．

IO 正則式（ストリーム式またはプロセス式）が決定性条件を満たすことは，既入力，既出力，現入力が同一のとき，次に選択される出力が一意に決まることを意味している．したがって，以下が成り立つ．

定理 7.1 独立な *IO* 正則式 t が決定性条件を満たせば，t は単調である．(証明略)

このことから，ある IO 正則式が決定性条件を満たせば単調であり，その意味集合を変えずに式を変換した場合でも，単調であることが保存されることになる．

プロセス式では，入力に依存しない出力のみを持つ基本項は無条件に実行され，その前で実行が打ち切られることはない．したがって，プロセス式では，入力チャネル上のストリームベクトルが同一なら，出力チャネル上のストリームベクトルも同一になる．従って，以下が成り立つことになる．

定理 7.2 独立なプロセス式 t が決定性条件を満たせば，t は関数的である．(証明略)

例題 7.2 チャネル *in*，*out* を含む *IO* 正則式 t は連接

$$t = t_1 t_2 = [in? \geq 0 \wedge out = in?]^*[out = 1]$$

であるとする．このとき実行条件集合は以下になる．

$$C(t_1) = \{(u \vdash a, v \vdash a) | u, v \in \Delta^* \wedge a \in \Delta_{(in)}\}$$
$$C(t_2) = \{(u \vdash, v \vdash 1) | u, v \in \Delta^*\}$$

これは　t_1, t_2が分離していないので，条件 *(4)* を満足しない．$\tau(t)$ には,$(u \vdash 2, v \vdash 21), (u \vdash, v \vdash 1)$ を共に要素として含み，単調ではない．

7.5　ストリーム代入の意味

プロセス式の入力チャネル変数の部分にストリームを当てはめ，プロセスからの出力ストリームを求めることを考える．このようなプロセス式の実行を，関数適用として，しかも全域的関数として扱うために，プロセス式 t の意味する集合$\tau(t)$　の入力チャネル上のストリーム集合（関数とした場合の domain）を$\vec{\Delta}'$　に拡張する．

定義 7.26 tをプロセス g のプロセス式とし，プロセス g の入力チャネル集合を *In*, 出力チャネル集合を *Out* とする．このとき，

$$\eta(t) = \tau(t) \cup \mu(t)$$

と定義する．ここに

$\mu(t)$ は次の条件を満たす $V \in \vec{\Delta}'$ の全体である．

適当な　$W \in \tau(t)$ に対して，

- $W_{(In)} \sqsubseteq V_{(In)}$ かつ $V_{(In)} \not\in \tau(t)_{(In)}$

- $W_{(Out)} = V_{(Out)}$

- $W_{(In)} \sqsubseteq Y_{(In)} \sqsubseteq V_{(In)}$かつ $Y \in \tau(t)$ なら
  $Y = W$

定義 7.27 ネットワーク全体のチャネル集合を Ch とおく. ストリーム式を s とし，入力チャネル集合 In，出力チャネル集合 Out を持つプロセスのプロセス式を p とする. $IN \cup Out \subset Ch$ であり，このプロセスの入力チャネル，出力チャネルのいずれでもないチャネルの集合を $Els = Ch - (IN \cup Out)$ とおく. このとき，ストリーム式の場合と同じ意味（$\tau(..)$）で，以下のストリームベクトル集合を意味する IO 正則式を，$p < s/(In) >$ で表し合成プロセス式と呼ぶ. 合成プロセス式を求めることを，プロセス式 p にストリーム式 s を代入するという.

$\tau(p < s/(In) >)$ は，以下を満たす $Y \in \vec{\Delta}'$ の集合

$Y_{(In)} = X_{(In)}, Y_{(Els)} = X_{(Els)}, Y_{(Out)} = P_{(Out)}$

(ただし X, P は、$X \in \tau(s)$, $P \in \eta(p)$ で $X_{(In)} = P_{(In)}$ を満たすとする)

したがって，入力のストリーム式をプロセス式に代入して，更に出力チャネル集合 Out 上の出力ストリーム式を求めることは，以下の集合を表す式を求めることである.

$$\tau(p < s/(In) >)_{(Out)}$$

このような代入は，IO 正則式の操作としては，以下のようにして行うことになる. いま，入力チャネル in の場合を例にとる.

入力ストリーム式を s，被代入プロセス式を t とする.

1. $\tau(s)_{(in)} \subset \tau(t)_{(in)}$ かつ，両者の式の上で チャネル名 in の出現の間に 1 : 1 対応が付けられるなら，すなわち，入力ストリーム式の in を含む基本項 $[p(in)]$ とプロセス式の in を含む基本項 $[g(in?)]$ が以下のように対応しているとき，

$$\frac{[p(in)]}{[g(in?)]}$$

次の論理関係を持った基本項が得られる. この操作が代入に相当する.

$$[p(in) \wedge g(in)]$$

この基本項は，一般の記号論理の規則によって式の変形を行うことができる.

2. 上記以外の場合. すなわち，$\tau(s)_{(in)} \subset \tau(t)_{(in)}$ を満たさないか，チャネル in の出現に 1 対 1 対応を直接見いだせない場合. ストリーム式の項の挿入位置を工夫するか，プロセス式の一部を除去するなどの方法で式を変形する.

被代入のプロセスの入力チャネル集合を In とすれば，代入は，ストリーム式とプロセス式で In の共通になっているストリームベクトルを両方から求め，プロセスの出力チャネル集合 Out の部分はプロセス式から，それ以外のチャネルの部分はストリーム式から要素をとり，ストリームベクトルを作るものである.

代入は，プロセスの入出力チャネルに関しては，IO 正則式の意味する集合 $\eta(p)$ の部分集合を求めることになる.

定理 7.3 ストリーム式 $s(Sin; Sout)$，プロセス式 $p(In; Out)$ の両方共に単調とする. 代入によって得られた合成プロセス式 $p < s/(In) >$ の入力チャネル集合を Sin として表現された式を $meet(Sin; Srm)$ とすれば、この式も，単調である. ただし，Srm は Sin 以外でこの合成式に表れるチャネル集合とする.

証明概略）代入前のストリーム式 s の意味集合の要素であるストリームベクトルを U, V とし，それらがプロセス式 p に代入された合成式の意味集合での対応する要素を $\dot{U}, \dot{V}$ とする. s が単調であるから，$U_{(Sin)} \sqsubseteq V_{(Sin)}$ と仮定すれば，$U_{(Sout)} \sqsubseteq V_{(Sout)}$ すなわち $U_{(In)} \sqsubseteq V_{(In)}$ が成り立つ. したがって，合成式の対応する要素の In の部分は，

$\dot{U}_{(In)} \sqsubseteq \dot{V}_{(In)}$が成り立つ．プロセス式が単調であることから$\dot{U}_{(Out)} \sqsubseteq \dot{V}_{(Out)}$も成り立ち，結局$\dot{U}_{(Srm)} \sqsubseteq \dot{V}_{(Srm)}$が成り立つ.(終)

例題 7.3 *IO* 正則式 s

$$s = [in = 1]^*[in = 0]$$

をプロセス式 p

$$p = [in? > 0 \wedge out = in?]^*[in? \leq 0]$$

に代入すると，$\tau(s)_{(in)} \subset \tau(p)_{(in)}$ より，

$$\{Y | Y_{(in)} = X_{(in)} \wedge Y_{(Els)} = X_{(Els)} \wedge Y_{(out)} = P_{(out)}$$

$$\wedge X \in \tau(s) \wedge \quad P \in \eta(p) \wedge X_{(in)} = P_{(in)}\}$$

$$= \tau([in = 1 \wedge in > 0 \wedge out = in]^*[in = 0 \wedge in \leq 0])$$

である (ただし，s, p いずれにも表れないチャネル集合を Els としている) から，

$$p < s/(in?) > \cong$$

$$[in = 1 \wedge in > 0 \wedge out = in]^*[in = 0 \wedge in \leq 0]$$

となる．次に，

$$r = [in = 3][in = 4]$$

を，同じプロセス式 p に代入すると，これも$\tau(r)_{(in)} \subset \tau(p)_{(in)}$ であるが，元の式のままではうまく対応付けはできない．そこで，プロセス式がその接頭部も意味集合に含むので，$\tau(p)$ がチャネル in から２つだけのアイテムを入力する部分が接頭部であることを考慮して，代入すれば

$$p < r/(in?) > \cong$$

$$[in = 3 \wedge in > 0 \wedge out = in]$$

$$[in = 4 \wedge in > 0 \wedge out = in]$$

となる．

プロセス p の入力チャネル全体を

$In = \{in_1, in_2, \ldots, in_k\}$, 出力チャネル全体を $Out = \{out_1, out_2, \ldots, out_m\}$ とし、

$$\vec{\Delta}'_{(In)} = \Delta'_{in_1} \times \Delta'_{in_2} \times \ldots \times \Delta'_{in_k}$$

$$\vec{\Delta}'_{(Out)} = \Delta'_{out_1} \times \Delta'_{out_2} \times \ldots \times \Delta'_{out_m}$$

$$\Psi = \{G | G \subset \vec{\Delta}'_{(In)}\}$$

$$\Phi = \{H | H \subset \vec{\Delta}'_{(Out)}\}$$

とおくと，

プロセス p のプロセス式 t を，代入と合成プロセス式から出力チャネルのストリームベクトル集合を求める操作と考えると，

$$t :: \Psi \to \Phi$$

なる型の関数であるといえる．プロセスの入力，出力チャネル以外のチャネル上のストリームは，代入によってそのまま置き換えられ，このプロセスへの代入操作には特別の意味を持たない．すなわち，プロセス式は入力チャネル上のストリームベクトル集合から，出力チャネル上のストリームベクトル集合への関数と考えられる．

なお，Kahn[22] では，プロセスを $\vec{\Delta}'_{(In)} \to \Delta'$なる型の連続関数として扱っている．一方，本稿のプロセス式は関数的であるから，単一の入力ストリームベクトルを代入すれば単一の出力ストリームベクトルを一意に決定する．すなわち，同じように，$\vec{\Delta}'_{(In)} \to \vec{\Delta}'_{(Out)}$ なる型の関数としても扱える．出力を各チャネル毎になるよう関数を分ければ各関数の型は、$\vec{\Delta}'_{(In)} \to \quad \Delta'_{out_i}$になる。従って，ストリーム式として，単一のストリームを表す表

現のみを用いて代入を進めていくことにすれば，Kahn[22] のモデルと同じであるといえる．しかし，本稿ではIO 正則式の代入を (ストリームベクトル集合を入力とする) 関数適用とするため，このような扱いにしている．

プロセス式の表す関数がΨ上の連続関数になることは，以下のようにして示すことができる．まず，

Ψ, Φは，$\sqsubseteq$ の関係の下で，cpo（complete partially ordered set:完備半順序集合）[44] になる．すなわち，

$$U_1 \sqsubseteq U_2 \sqsubseteq \ldots \sqsubseteq U_i \sqsubseteq \ldots$$

$$(\text{ただし}, U_i \in \Psi \quad (i = 1, 2, \ldots))$$

に対し，

$$\lim_{i \to \prime} U_i \in \Psi$$

も成り立つ（Φ　についても同様)．

プロセス式tに対し，$\mu(t)$ は全域関数であり，プロセス式では，L^*, L^ωが共に　$\lim_{k \to \infty} \tau(L^k)$ を包含している．いま，

$$U_1 \sqsubseteq U_2 \sqsubseteq \ldots \sqsubseteq U_i \sqsubseteq \ldots$$

を満たす　U_iをプロセスに代入することを考える．この列に対し，$\forall k > 0 : \exists j > k : U_j \neq U_k$が成り立つなら，実質的な無限上昇列である．

$$T_{1(In)} = U_1 \wedge T_{1(Out)} = p(U_1),$$

$$T_{2(In)} = U_1 \wedge T_{2(Out)} = p(U_1)$$

$$\vdots$$

であり，$T_i \subset \tau(p)$ なる $T_1, T_2, \ldots, T_i, \ldots$ が存在する．

このようなT_i は，プロセス式の*やωの項に対応するので，

$$T_\infty = \lim_{j \to \infty} T_j$$

なるT_∞が$\tau(t)$ に含まれる．すなわち，プロセス式tを関数として扱うと，

$$t(\lim_{i \to \infty} U_i) = t(T_{\infty(In)}) = T_{\infty(Out)}$$

かつ，

$$\lim_{i \to \infty} t(U_i) = \lim_{i \to \infty} t(T_{i(In)})$$

$$= \lim_{i \to \infty} T_{i(Out)} = \lim_{i \to \infty} T_{\infty(Out)}$$

が成り立つ．すなわち，

$$t(\lim_{i \to \infty} U_i) = \lim_{i \to \infty} t(U_i)$$

である．一方

$$U_1 \sqsubseteq U_2 \sqsubseteq \ldots \sqsubseteq U_i \sqsubseteq \ldots$$

が，途中のU_iからすべて同一になる，すなわち

$$\exists k > 0 : \forall j > k : U_j = U_k$$

を満たすなら，やはり

$$t(\lim_{i \to \infty} U_i) = \lim_{i \to \infty} t(U_i)$$

が成り立つ．すなわち，関数tは連続関数である．cpo 上の連続関数は、最小不動点を持つことは知られている [44][22]。データフローネットワークの各プロセス式は連続関数になり (個々の出力チャネルを値域とする関数に分けることができる)、従って、ストリームベクトル集合として，ネットワークの解である最小の不動点を持つことになる [22]。このことが、以降の検証の裏付けになる。

7.6　ネットワークの検証

ネットワークの実行開始は，以下のいずれか又は両方によって行われる．

1. 外部入力チャネルからのデータをあるプロセスが入力する．これを外部入力ストリームという．外部入力ストリームはネットワーク実行開始時にチャネル上に一括して与えられているとして扱う．

2. あるプロセスが入力チャネルからの入力なしに，出力チャネルにデータを出力する．これを，初期出力データという．このようなプロセスのみが，入力チャネル上の空入力に対して，出力チャネル上に空でない出力を出すことが出来る．

ネットワークの各プロセスのＩＯ正則式が得られており，各プロセスについてはいずれもその正当性が検証済みであるとする．個々のプロセス式はそれぞれの入力出力ストリーム間の関係を表現しており，当初は外部入力データや初期出力データとの関係が付けられていない．検証する特定のプロセスにおける出力ストリームと，外部入力データ，初期出力データとの関係をつけ，IO 正則式で表す．これが検証したい出力ストリームの満たすべき条件を表したもの（すなわち仕様）であり，目的式と呼ぶ．

例えば外部入力チャネル $(in_1?, in_2?, \ldots, in_k?)$，外部出力チャネル (out_1, out_2) をもつネットワーク（以下ネットワークＮとよぶ）の場合，チャネル out_1, out_2に対する目的式 f_1, f_2（の表す集合）は，それぞれ以下のようになる．

$$\tau(f_1) \subset \Delta'_{in_1} \times \Delta'_{in_2} \times \ldots \times \quad \Delta'_{in_k} \times \Delta'_{out_1}$$

$$\tau(f_2) \subset \Delta'_{in_1} \times \Delta'_{in_2} \times \ldots \times \quad \Delta'_{in_k} \times \Delta'_{out_2}$$

尚，我々の方法では，仕様を目的式（IO 正則式）で表現するが，仕様が一般的な論理式で表現されていた場合，目的式がこの仕様を満足することを，Hoare 論理 [5, 17] などによって示すことは可能である．すなわち，目的式は仕様を間接的に証明するための中間的な表現であるということができる．

ネットワークの実行が永久に続けられたときの究極の状態をチャネル上のストリームに着目して考えると，以下の２つになる．

1. すべてのチャネルにおいて，出力ストリーム全体が入力ストリームとしていずれかのプロセスによって入力された状態 (すなわち，無限長のストリームもすべて入力し尽くされる状態があるとみなせば，チャネル上に入力残のストリームがない状態)
2. チャネル上に出力されたまま永久に入力されないデータが残されている状態．

上記のいずれかの状態でネットワークの実行を終了した時点で (または永久に実行を継続して)，この目的式を満たすこと，が検証すべきことである．

つぎに，各プロセスの入力チャネルへの入力ストリームを IO 正則式で表す．これを，入力正則式と呼ぶ．入力正則式については，その集合がプロセス式 p の意味集合$\tau(p)$ のドメインに包含されている場合（すなわち上記 (1)）と，包含されない（もちろん$\eta(p)$ には包含される）場合（すなわち上記 (2)）とがある．

例えばネットワークＮで，プロセス $p1$ が入力チャネル $(ich_1?, ich_2?)$ を持つとすれば，$ich_1?, ich_2?$に対する入力ストリームの IO 正則式 g_1, g_2は，上記 (1) では，以下を満たす．

$$\tau(g_1)_{(ich_1)} \subset \tau(p1)_{(ich_1)}, \tau(g_2)_{(ich_2)} \subset \tau(p1)_{(ich_2)}$$

上記 (1),(2) いずれの状態でも，以下を満たす．

$$\tau(g_1)_{(ich_1)} \subset \eta(p1)_{(ich_1)}, \tau(g_2)_{(ich_2)} \subset \eta(p1)_{(ich_2)}$$

次に入力正則式をプロセス式に代入する．入力正則式がプロセス式 p の意味集合$\tau(p)$ のドメインに包含されている場合（すなわち上記 (1) 状態）は，プロセス式 p の入力チャネルの条件の部分に更に条件を付加することによってこの代入がなされる．上記状態 (2) の場合は，プロセスによって入力されない部分も入力正則式の後部に含まれる可能性があり，プロセス式にも対応する入力データがない部分も含まれる可能性があり，それらの部分をそれぞれから除去したうえで，当てはめた式を作らなければならない．このようにして得られたストリーム式が合成プロセス式である．上記例の入力正則式 g_1, g_2をプロセス式 $p1$ に代入した結果の合成プロセス式は以下で表される．

$$p1 < g_1/(ich_1?), g_2/(ich_2?) >$$

こうして，合成プロセス式にはそのプロセスに属さないチャネルの記述も含まれることになる．すべてのプロセスについても同様に，プロセス式に入力正則式を代入し，合成プロセス式を構成する．

こうして得られた合成プロセス式から，プロセスの出力チャネル上の出力ストリームをIO正則式の形で求める．これを，出力正則式とよぶ．例えば出力チャネル och_1, och_2 に対する出力正則式をそれぞれ以下のように表す．

$$p1 < g_1/(ich_1?), g_2/(ich_2?) >_{[och_1]}$$

$$p1 < g_1/(ich_1?), g_2/(ich_2?) >_{[och_2]}$$

こうして，すべてのチャネルに対し，出力としてのストリーム集合（出力正則式）が入力としてのストリーム集合（入力正則式）に等しいことを示せば，得られた出力正則式のストリームベクトルの集合が，このネットワークの不動点であることになる．

更に，全チャネルの表すストリームベクトルが最小の不動点になることを示す必要がある．最後に，得られたストリームベクトルの集合から目的式が証明されれば，検証が完了する．

なお，この方法によって証明される結果が正しいものであることは，概略を以下のように説明することができる．

ストリーム集合の半順序関係の下で，プロセス式による代入は，ストリームベクトル集合からストリームベクトル集合への連続関数 [22] であるので，入力ストリーム集合から出力ストリーム集合を求めること (ア) は，連続関数の関数値を求めることに相当する．次に，すべてのチャネル上の入力側，出力側のストリーム集合の一致が示される (イ)．故に，上で得られたストリームのベクトル集合がこのネットワークの不動点になる．最小の不動点であることが示されれば (ウ)，Kahn[22] によってこのストリームベクトル集合が唯一の解であることが保証される．最後は，このストリームベクトル集合から，仕様が証明される (エ)．

入力ストリームを与える部分などは発見的であるとはいえ，検証過程のうち，(ア),(イ),(ウ) の部分の証明が正しければ，ネットワークを満たすストリームベクトル集合が必ず得られる．従って，さいごの (エ) の証明が正しいことが保証されれば，全体の検証が正しいこ

とになる．ただし，(ア),(イ),(ウ),(エ) の段階の具体的証明手段は，我々の検証方法の範疇には含まれていないので，枠組みとして正しい結果を得るための方法であるといえる．

検証の手順は以下のように要約することができる．

1. 外部入力があれば，入力正則式として表現する
2. 目的とするチャネル上の出力ストリームの条件（目的式）を，IO正則式として表現する
3. 各プロセスの入力チャネルの入力正則式を作る．
4. 各プロセス式にこの入力正則式を代入し合成プロセス式とする．
5. 各プロセスについて，合成プロセス式から出力チャネルの出力正則式を作る
6. 必要なら，各チャネルについて，出力正則式　≅入力正則式　を満たすことを証明する
7. 各チャネル上の出力正則式の表すストリームが最小不動点であることを示す．
8. 目的式が，前段階までに導出された各ストリームのIO正則式から導かれることを証明する．

7.7　ネットワークに対する検証の例

Kahn[22] が例としてあげているものを，やや簡単化してとりあげ我々の方法によって検証を行う．

例題 7.4　(フィードバックを含む例)

下図で，プロセス g は最初チャネル y,z にそれぞれ *0,1* を出力し，以降はチャネル x からの入力を y,z に交互に出力し続ける．一方，プロセス f は，y,z を交互に入力しそれを x に送出し続ける．このデータフローネットワークにおいて，チャネル x のストリームは，*1,0* が交互に現れる無限列であることを証明する．

以下，前節の手順番号を併記して検証を進めていく．

手順 *(2),(3)*：x 上のストリーム（目的式）

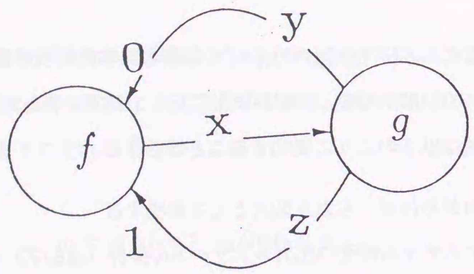


図 1: フィードバックを含むプロセス間通信

$$fin(x) = [\quad [x=0]\,[x=1]]^{\omega}$$

*(*ストリーム式であるから，ωは無限列*)*

交互送信プロセス g のチャネル

入力チャネル	*(x)*
出力チャネル	*(y,z)*

プロセス g のＩＯ正則式

$$g(x;y,z) = [y=0 \wedge z=1]\,[[y=x?]\,[z=x?]]^{\omega}$$

すなわち，プロセス g は，入力チャネル x からの入力と，出力チャネル *y,z* への出力を永久に続ける可能性がある *(*プロセス式だからωは，有限で終了する場合も含む*)* ものである．

手順 *(4)*：プロセス g の合成プロセス式

$$g < fin(x)/(x?) > \cong$$

$$[y=0 \wedge z=1]\,[[y=0]\,[z=1]]^{\omega}$$

手順 *(5)*：出力 y のストリーム

$$g < fin(x)/(x?) >_{[y]} \cong [y=0]\,[y=0]^{\omega}$$

手順 *(5)*：出力 z のストリーム

$$g < fin(x)/(x?) >_{[z]} \cong [z=1]\,[z=1]^{\omega}$$

交互受信プロセス f のチャネル

入力チャネル	*(y,z)*
出力チャネル	*(x)*

プロセス f のＩＯ正則式

$$f(y,z;x) = [[x=y?]\,[x=z?]]^{\omega}$$

手順 *(4)*：プロセス f の合成プロセス式

$$f < g_{[y]}/y?, g_{[z]}/(z?) > \cong$$

$$[x=0]\,[x=1]\,[[x=0]\,[x=1]]^{\omega}$$

手順 *(5)*：出力 x のストリーム

$$f < g_{[y]}/(y?), g_{[z]}/(z?) >_{[x]} \cong$$

$$[x=0]\,[x=1]\,[[x=0]\,[x=1]]^{\omega}$$

手順 *(6)*：入力，出力ストリーム式の意味集合が一致すること．

$$f < g_{[y]}/(y?), g_{[z]}/(z?) >_{[x]},$$

$$fin(x)$$

これらは，どちらも *01* の無限列である．故に，

$$f < g_{[y]}/(y?), g_{[z]}/(z?) >_{[x]} \cong fin(x)$$

が成り立つ.

更に，$(f < g_{[y]}/(y?), g_{[z]}/(z?) >_{[x]}$ ， $g < fin(x)/(x?) >_{[y]}, g < fin(x)/(x?) >_{[z]})$ が最小不動点であることも以下のように示すことができる.

手順 *(7)*：$f_{[x]} = (01)^{\omega}$の真の接頭部が不動点になっていると仮定．すなわち，適当な非負整数 k に対し，$f_{[x]} = (01)^{k}$．これを gの入力ストリームとすれば，出力 y,zのストリームとして両方合わせて最大長さ k *+2*のストリームが得られ，再度 fの入力とすれば，fからの出力ストリームは最大長さ k*+2*になる．すなわち，$(01)^{k}$は不動点ではない．他のチャネルについても同様に示される．よって証明が完了.

例題 7.5 *(*分散アルゴリズム *(*最大値発見問題*) [40]*の正当性検証の例*)*

n 個のプロセスがリング状にチャネルで結ばれているネットワークとし，プロセスを $P_0, P_1, \ldots, P_{n-1}$，各プロセス P_iは入力チャネル $x_{md(i-1)}$, 出力チャネル x_iを持つとする (但し，$md(k) = k \bmod n$ とおく)．また各プロセス P_iは独自の異なる識別値 val_iを持つとする．すべてのプロセスが同一アルゴリズムでデータをチャネルを通して送受信しながら，識別値の最大値をすべてのプロセスが知るようにする，というのが問題である．アルゴリズムとしては，最も簡単な方法を対象とする．すなわち，ある有限時間内にすべてのプロセスは実行を開始するものとし，各プロセス P_iは，まず自分の識別値を出力して，以降は再度自分の識別値を入力するまで，すべての入力をそのまま出力しつづける．その間に局所チャネル mxに最大値を記憶していくものとする.

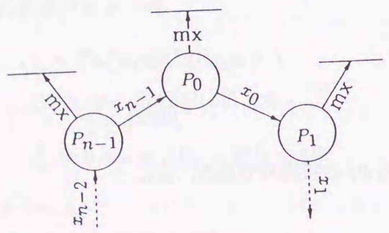


図 2: 最大値発見問題

プロセス P_iのチャネル.

入力チャネル	$(x_{md(i-1)})$
出力チャネル	(x_i)
局所チャネル	(mx)

以下で $c\natural$は，ストリームの補助変数とする（同一名でも，個々の *IO* 正則式毎に異なる変数を表す）.

プロセス式 P_i：

$$\left[\begin{array}{l} x_i = val_i \\ \wedge mx = val_i \end{array}\right] \left[\begin{array}{l} x_{md(i-1)}? \neq val_i \\ \wedge mx = max(\overline{mx}, x_{md(i-1)}?) \\ \wedge x_i = x_{md(i-1)}? \end{array}\right]^{*}$$

$$\left[x_{md(i-1)}? = val_i\right]$$

目的式：局所チャネル mxに最大値が得られることを検証する．任意のプロセス P_iに対し，mxが以下のストリームなることを示すことが必要である.

$$\left[\begin{array}{l} c\natural = i \\ \wedge mx = val_i \end{array}\right] \left[\begin{array}{l} c\natural \neq i \\ \wedge mx = max\{val_k | \\ \quad k \in wch(c\natural, i)\} \\ \wedge c\natural = md(\overline{c\natural} - 1) \end{array}\right]^{*}$$

なお $wch(a,b)$ は以下の集合とする.

$$wch(a,b) = \{a = md(a), md(a+1), \ldots, md(a+k) = b\}$$

(但し,$0 \leq k < n$)

プロセス P_iの入力チャネル $x_{md(i-1)}$へのストリームを

$$[c\natural = i] \left[\begin{array}{l} x_{md(i-1)} \neq val_i \\ \wedge x_{md(i-1)} = val_{c\natural} \\ \wedge c\natural = md(\overline{c\natural} - 1) \end{array}\right]^{*} \left[x_{md(i-1)} = val_i\right]$$

と仮定し，プロセス式 P_iに代入すれば,

$$\left[\begin{array}{l} c\natural = i \\ \wedge x_i = val_i \\ \wedge mx = val_i \end{array}\right] \left[\begin{array}{l} x_{md(i-1)}? \neq val_i \\ \wedge x_{md(i-1)}? = val_{c\natural} \\ \wedge c\natural = md(\overline{c\natural} - 1) \\ \wedge mx = max(\overline{mx}, val_{c\natural}) \\ \wedge x_i = x_{md(i-1)}? \end{array}\right]^{*}$$

$$\left[x_{md(i-1)}? = val_i\right]$$

が得られる．この式より $x_{md(i-1)}?$を消去して，チャネル x_i上のストリームを求めれば，

$$[c\natural = md(i+1)]\left[\begin{array}{l} x_i \neq val_{md(i+1)} \\ \wedge x_i = val_{c\natural} \\ \wedge c\natural = md(\overline{c\natural} - 1) \end{array}\right]^*$$

$$\left[x_i = val_{md(i+1)}\right]$$

となり，これはプロセス $P_{md(i+1)}$への入力ストリームとしてプロセス P_iの入力ストリームと（添字を除いて）全く同じ形をしている．したがって，同じようにプロセス $P_{md(i+1)}$に代入することができる．

最小不動点になること：

仮定したストリームの真の接頭語が解であるとすれば，それが一巡したとき，更に一つ識別値を加えたものが入力ストリームとなる *(矛盾)*．したがって本検証で与えたストリームが最小不動点であることになる．

局所チャネル mx 上のストリームを求めると，

$$\left[\begin{array}{l} c\natural = i \\ \wedge mx = val_i \end{array}\right]\left[\begin{array}{l} c\natural \neq i \\ \wedge mx = max(\overline{mx}, val_{c\natural}) \\ \wedge c\natural = md(\overline{c\natural} + 1) \end{array}\right]^*$$

この式が目的式を満たすことは帰納法で証明することができる．こうして，証明が終了する．

7.8 IO 正則式による検証の評価

本節では、データフローネットワークにおけるストリームとプロセスの表現（IO 正則式）と，その表現の下でのネットワークの検証方法論を提案した．IO 正則式を構成する基本項は，データ間の静的な論理式から成り，その静的論理式を連接，選択，反復の演算で結ぶことによって，データの動的な出現関係を表現している．データフローネットワークに表れるような，入力ストリームと出力ストリームの関係を証明する問題では，本節で提案した表現法と証明手法が有効であろうと考えている。

本節で述べた方法で検証を進める上での難しさが少なからず存在する。検証の手順においては，プロセス式にストリームを代入する段階，チャネル上の入力，出力ストリームが同一なることの証明段階，の２つがポイントである．前者に対しては，代入すべきストリームの式の作成基準，プロセス式をそのストリームの式に整合させる方法など，現段階ではそれらの一般的方法を見いだすに至っていない．後者について，我々は等価な IO 正則式を導出するための公理, 推論規則を作りつつあるが，完全なものには至っていない．

そのほか，ストリーム式を見いだすことを発見的に行う必要があること，複数の入力チャネルの場合に，複数のチャネルをまとめたストリーム式にする必要が生ずる場合があること，IO 正則式で表された目的式から一般的な論理式で表された仕様を証明する統一的方法を示していないこと，などが挙げられる．

8　おわりに

IO 正則式に基づく仕様表現による洗練化，検証の方法を提案した。IO 正則式はコンピュータプログラムの静的な論理関係の部分と動的な状態推移の部分を分離して表現する表記法である。仕様，設計いずれにおける表現としても使うことができ、そのことが，システム開発を一貫した表現法の下に進めることができる点で有効であると考えられる。仕様段階に対して言えば、特に外部環境との相互作用があるシステムは、入力出力に伴うシステム状態の推移を，ストリームや列などのデータ構造によって表現することが困難であり、このような場合には、IO 正則式によって、個々の入出力データ間の関係によって仕様を記述することが有効である。そのことは、データフローネットワークの場合に顕著に現れる。データフローネットワークにおけるプロセスは、ストリームをストリームに変換する機構と考えられるが、直接ストリーム間の関係として表現する事が容易ではなく、またそのようにストリーム関係で表現しても、設計への移行がスムーズではない。IO 正則式は，最初データフローネットワークの計算モデルとして提案されたものである。

IO 正則式の基本項には，Z のスキーマが適合し、本論文では，Z に基づいた IO 正則式が仕様，設計の記法として提案され、考察された。従来の洗練化に対し、洗練化過程も IO 正則式という統一した記法で進めることができる点がすぐれていると考えている。ただし、IO 正則式には、並行処理の記述がなく、並行処理を伴うシステムに対しては、CSP や CCS などの表現法に Z スキーマを取り込む方法がより有効である。しかし、IO 正則式に並行処理の機能を加えることは可能であろうと思われる。その点に対する拡張が課題として残されている。

本論文のプログラミング言語として, 反復を while 文に限定し，関数や手続き宣言を扱わないものに限定して述べてきた。しかし, より現実のプログラムに対応させる為に,for 文や repeat 文, 再帰を含む関数宣言などを初めとする, プログラム言語の持つ種々の表現への対応も考えていかねばならない。しかし、そのような拡張の下でも基本的な考えは変わらないと考えている。

本論文では、さらに決定性データフローネットワークを形式化し、ヒストリとしての最小不動点の存在を示した。我々のモデルでは、ネットワークの計算するヒストリ関数は連続関数になること、すなわち、任意の決定性データフローネットワークが最小不動点をもつことが判った。さらに、この不動点に対し操作的意味を対応させることができた。すなわちネットワークのすべての内部チャネルが空になるか、あるいはチャネル上のデータをどの基本プロセスも入力出来ない状態に達するなら、有限長の不動点が存在するということができる。

本論文の決定性データフローネットワークが Kahn のプログラミング言語[22]や Stella[41]の基本機能を包含していることを示すことができるが、本論文ではほとんど触れなかった。しかし一般にプログラミング言語が、入力チャネルが空か否か（すなわちチャネルにデータが到着しているか否か）を判定する機能を含んでいれば、入力, 出力ヒストリの関係は、チャネル上にデータが発生するタイミングに依存する。すなわち、同一入力のヒストリに対し、複数の出力ヒストリが対応し、その関係は本論文における基本プロセスのような部分関数にはならない。今後は、そのような並列処理言語と同等な機能をもつ決定性 (ヒストリとしては非決定性) のネットワークのもとでの意味論を考察したいと考えている。

さらに、データフローネットワークにおけるストリームとプロセスの表現（IO 正則式）と，その表現の下でのネットワークの検証方法論を提案した。IO 正則式を構成する基本項は，データ間の静的な論理式から成り，その静的論理式を連接，選択，反復の演算で結ぶことによって，データの動的な出現関係を表現している。データフローネットワークに表れるような，入力ストリームと出力ストリームの関係を証明する問題では，本節で提案した表現法と証明手法が有効であろうと考えている。ただし，プロセスの入出力に構造不一致がある場合にはプロセスを詳細化し、構造一致になるよう変換する必要がある。

現段階では、本稿の方法で検証を進める上での難しさが少なからず存在する。検証の手順においては，プロセス式にストリームを代入する段階，チャネル上の入力，出力ス

トリームが同一なることの証明段階，の２つがポイントである。前者に対しては，代入すべきストリームの式の作成基準，プロセス式をそのストリームの式に整合させる方法など，現段階ではそれらの一般的方法を見いだすに至っていない。

そのほか，ストリーム式を見いだすことを発見的に行う必要があること，複数の入力チャネルの場合に，複数のチャネルをまとめたストリーム式にする必要が生ずる場合があること，IO 正則式で表された目的式から一般的な論理式で表された仕様を証明する統一的方法を示していないこと，などが挙げられる．

謝辞

本論文に至るまでの研究の過程で、北海道大学宮本衛市教授には、親身なご指導とご教示を賜り、時には御討論を戴きました。深く感謝の意を表します。

北海道大学 嘉数侑昇教授、同大学 大内東教授、同大学 和田充雄教授には，論文の審査にあたり、有益な御討論と助言を戴きました。厚く御礼を申し上げます。

長年にわたって、室蘭工業大学 山口忠教授と札幌学院大学 長田博泰教授に、研究に関して議論をして戴いたことが、この論文をまとめる原動力になっていたと感じています。ここに感謝致します。

北海道情報大学の三枝武男前学長は、常に励ましを与えて下さいました。さらに、北海道情報大学 大野公男学長はじめ、本学の多くの教員の方々の援助があってこそ、現在まで研究を進めることができたものと思います。これらの方々に御礼を申し上げます。

参考文献

[1] Back,R.J. and Wright,J.: Refinement Calculus, Springer(1998).

[2] Bird,R. and Wadler,P.: Introduction to Functional Programming, Prentice-Hall, 1988 (武市 正人訳: 関数プログラミング, 近代科学社, 1991)

[3] Brock,J. and Ackerman,W.:"Scenarios: A Model of Nondeterminate Computation", Lecture Notes in Computer Science 107,Springer-Verlag,pp.252-259(1981)

[4] Brooks.Jr,F.P.: No Silver Bullet-Essence and Accidents of Software Eingineering, IEEE Computer, vol.20, no.4, pp.10-19, 1987.

[5] Dahl,O.: Verifiable Programming, Prentice Hall, 1992.

[6] DeMarco,T.: "Structured Analysis and System Specification,"YOURDON,1979 邦訳：高梨智弘, 黒田純一郎訳，"構造化分析とシステム仕様，"日経 BP 社,1986

[7] Dijkstra,E.W.: A Method of Programming, Addison-Wesley, 1988. (玉井浩訳: プログラミングの方法, サイエンス社, 1991)

[8] Diller,A.: Z - An Introduction to Formal Methods. John Willey & Sons, 1994.

[9] Duke,R., Hayes,P.K. and Rose,G.A.: Protocol Specification and Verification Using Z, in Protocol Specification, Testing, and Verification **VIII** (Aggarwal,S. and Sabnani,K. editors), Elsevier Science Publishers (North-Holland), pp.33-46, 1988.

[10] Duke,R., Gorden,R. and Smith,G.: Object-Z: A Specification Language Advocated for the Description of Standards, Computer Standards & Interfaces, **17** , pp. 511–533, 1995.

[11] Evans,A.S.: An Improved Recipe for Specifying Reactive Systems in Z, Proceedings of ZUM'97 (Lecture Notes in Computer Science, vol.1212), Springer-Verlag, pp.275–294, 1997.

[12] Fisher,C.: How to Combine Z with a Process Algebra, Proceedings of ZUM'98 (Lecture Notes in Computer Science, vol.1493), Springer-Verlag, pp.5–23, 1998.

[13] Fisher,C.: CSP-OZ:A Combination of Object-Z and CSP, Formal Methods for Open Object-Based Distributed Systems (FMOODS'97),**2**, pp.423-438, 1997.

[14] Futamura,Y., Kawai,T., Horikoshi,H. and Tsutsumi,M.: Development of Computer Programs by Problem Analysis Diagram, Proceedings of 5th International Conference on Software Engineering, IEEE, pp.325-332, 1981.

[15] Gries,D.: The Science of Programming, Springer-Verlag, 1981.

[16] Hayashi,Y., Yamaguchi,T. and Miyamoto,E.: A Proof Framework with IO Regular Expressions for Dataflow Networks, Work-in-progress papers of IRW/FMP'98, The Australian National University, TR-CS-98-09, pp.22-39, 1998.

[17] Hoare,C.A.R.: An Axiomatic Basis for Computer Programming, Comm.ACM **12**, no.10, pp. 576–580, 1969.

[18] Hoare,C.A.R.: Communicating Sequential Processes, Prentice-Hall, 1985.

[19] Hoare,C.A.R.: How did Software Get so Reliable without Proof?, Lecture Notes in Computer Science, vol.1051, pp.1-17, 1996.

[20] Hughes,J.W.: A Formalization and Explication of the Michael Jackson Method of Program Design, Software Practice and Experience, vol.9, pp.191-202, 1979.

[21] Jackson,M.A.: Principles of Program Design,Academic Press inc., 1976. (鳥居宏次訳：構造的プログラム設計の原理, 日本コンピュータ協会,1980)

[22] Kahn,G. :"The Semantics of a Simple Language for Parallel Programming," Information Processing 74:Proceeding of IFIP74, North-holland,pp.471-475,1974

[23] Kahn,G. and MacQueen,D.B.:"Coroutines and Networks of Parallel Processes,"IFIP77,pp.993-998,North-Holland,1977

[24] Keller,R.M.:"Denotational Models for Parallel Programs with Indeterminate Operators,"in Formal Description of Programming Concepts(Neuhold,E.J. ed.),North Holland,pp.337-366,1977

[25] Kok,J.N.:"Denotational Semantics of Nets with Nondeterminism," Lecture Notes in Computer Science (Springer-Verlag),vol.206,pp.237-249,1988

[26] Kok,J.N.:" An Iterative Metric Fully Abstract Semantics for Non deterministic Dataflow ", Lecture Notes in Computer Science 379 , Springer-Verlag, pp.321-330 (1989)

[27] Kowalski,R.: Algorithm= Logic + Control, Comm. ACM, vol.22, no.7, pp.424-436, 1979.

[28] Jones,C.B.: Systematic Software Development using VDM, Prentice-Hall, 1990.

[29] Lynch,N.A. and Stark,E.W.:"A Proof of the Kahn Principle for Input/Output Automata",Information and Computation,vol 82,pp.81-92(1989)

[30] R.Milner: Communication and Concurrency, Prentice-Hall, 1989.

[31] Morgan,C.:Programming from Specifications, Prentice-Hall, 1994.

[32] Nguyen,V.,Demers,A.,Gries,D.and Owicki,S.: "A Model and Temporal Proof System for Networks of Processes," Distributed Computing,vol.1,pp.7-25,1986

[33] Potter,B.,Sinclair,J. and Till,D: An Introduction to Formal Specification and Z, Prentice-Hall, 1991. (田中　武監訳：ソフトウエア仕様記述の先進技法－Z言語, トッパン,1993).

[34] Roscoe,A.W. Woodcock,J.C.P. and Wulf,L.: Non-Interference through Determinism. Proceedings of ESORICS 94, (Lecture Notes in Computer Science, vol.875) ,Springer-Verlag pp.33–54, 1994.

[35] Stoddat,W.J.: An Introduction to the Event Calculus, Proceedings of ZUM'97 (Lecture Notes in Computer Science, vol.1212) ,Springer-Verlag, pp.10–36, 1997.

[36] Taguchi,K. and Araki,K.: Extending Z with State-Transition Constraints, Proceedings of International Conference of Computer Software and Applications (COMPSAC),pp.254-260, 1996.

[37] Taguchi,K. and Araki,K.: The State-Based CCS Semantics for Concurrent Z Specification, International Conference of Formal Engineering Methods(ICFEM), IEEE Computer Society Press, pp.283–292, 1997.

[38] Wirth,N.: Algorithms + Data Structures = Program, Prentice-Hall, 1976. (片山卓也訳: アルゴリズム + データ構造 = プログラム, 日本コンピュータ協会, 1979)

[39] Woodcock,J.C.P. and Davies,J. : Using Z Specification, Refinement, and Proof, Prentice Hall, 1996.

[40] 亀田恒彦，山下雅史：分散アルゴリズム, 近代科学社,1994

[41] 久世和資，佐々政孝，中田育男：ストリームによるプログラミングのための言語とその実現方式，情報処理,vol.31,No.5,pp.673-685,1990

[42] 小林　孝次郎, 高橋　正子:オートマトンの理論, 共立出版,1983

[43] 田中二郎：関数型プログラムにおけるストリーム計算, 情報処理，vol.29,no.8,pp.836-844,1988

[44] 横内　寛文：プログラム意味論, 共立出版，1994

[45] 林　晋: プログラム検証論. 共立出版 ,1995.

[46] 林　雄二:　ジャクソン法における構造不一致の形式化,　電子情報通信学会論文誌 D-**I**, vol.J76, no.1, pp.11-18,1993.

[47] 林　雄二:　部分関数を定義する m-n 列変換器の性質, 電子情報通信学会論文誌D-**I**,vol.J77, no.6, pp.409-414,1994.

[48] 林　雄二:　決定性データフローネットワークにおける不動点の意味, 電子情報通信学会論文誌 D-**I**, vol.J78, no.9, pp.820-830,1995.

[49] 林　雄二：データフローネットワークを基礎にしたプログラミング言語の開発, 信学技報, SS95-10, pp.1-8, 1995

[50] 林　雄二, 山口　忠:　ＩＯ正則表現によるデータフローネットワークの検証, 電子情報通信学会論文誌 D-**I**, vol.J81, no.6, pp.755-769, 1998.

[51] 林　雄二：Ｚを基にした正則表現による仕様記述と検証, ソフトウエア工学の基礎VI (日本ソフトウエア科学会 FOSE'99),pp.156-163, 近代科学社,1999

[52] 松本　正雄, 小山田　正史, 松尾谷　徹: ソフトウエア開発・検証技法, 電子情報通信学会編, 1997.