ICOT Technical Memorandom: TM-1024

TM-1024

並列論理型言語の形式的意味論

村上　昌己（富士通）

February, 1991

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

解説

並列論理型言語の形式的意味論[1]

村上 昌己[2]

1 はじめに

並列論理型言語 (Concurrent Logic Languages) はその名の通り論理型プログラミング言語の枠組みから発展した言語であり、並列 / 分散処理システムの基礎 / 応用ソフトウェアの記述言語として提案されてきた。これらの言語は、ソフトウェア中での並列性 (concurrency) の記述に必要なプロセスおよびそれらの間の通信 / 同期 / 制御の機能を論理型言語の枠組みで実現するために、独特の言語機能とプログラミングの作法を採用している。

具体的な言語としては PARLOG [34, 14], Concurrent Prolog [66, 67], GHC [71] 等がよく知られているが、これらの言語は大体において言語の基本的なシンタックスやプログラミングの技法に共通する部分が多く、実際はその他の類似の言語も含めてコミット選択型並列論理型言語 (Committed choice concurrent logic languages) と呼ばれ、ひとまとまりに扱われることも多い [68]。

これらの言語の設計思想、特にプロセス / 同期 / 通信の記述のための手法及び機能については、[24, 74, 72, 73] に詳しく述べられているが、簡単に言えば次のようになる。並列論理型言語による並列プログラミングの基本的な作法では、プロセスの並列実行を各プロセスを表現するゴールの論理積によってあらわし、複数のプロセスの間の通信をそれらのゴールに共有されている変数の単一化によってあらわす。プロデューサとなるプロセスが共有変数を (部分的にでも) 具体化すると、その束縛情報はプロセスの終了を待つことなくその変数を共有するコンシューマのプロセスに伝えられ、両方のプロセスは並列に実行が進められる (このような並列処理の方法をストリーム and 並列と呼ぶ)。コミット選択型と呼ばれる言語の共通の特徴は、このようなプロセスの同期 / 制御の方法として、各プロセスにとって「受動的に解かれるべき部分」と「能動的に解くことができる部分」を区別するための機構を純 Prolog につけ加えた形で言語が定義されているという点である。

---

[1] Formal Semantics of Concurrent Logic Programs by Masaki MURAKAMI (International Institute for Advanced Study of Social Information Science, FUJITSU LIMITED).

[2] 富士通 (株) 国際情報社会科学研究所

当初このような言語のセマンティクスは、処理系の設計 / 実現の指針を与えるものとして、自然言語によって非形式的に記述されていた。しかしながら、その後様々な形式的手法による意味論、すなわちプログラムの意味を何らかの数学的領域の元として与えるセマンティクスが提案されている。

これらの言語の分野での研究分野として形式的意味論の研究の重要性が認識されはじめた背景としては、第一に論理型言語 (特に純 Prolog) の分野でプログラムの検証 / 変換 / 合成の手法等、論理型言語の「理論的明瞭さ」を生かすような研究が様々な形で進められたことがあげられる。並列論理型言語の場合も純 Prolog 等での研究の成果を受けるような形で、プログラムの検証 / 変換 / 合成 / 抽象解釈の手法等の研究に取り組む動きが出てきた [50, 23, 75, 16]。その結果、それらの研究の基礎となる言語の形式的意味論が注目されるようになったという次第である。

また一方で CCS[49]、 CSP [35] 等並列計算モデルの理論の研究が進められ、それらの形式的意味論についての結果 [3, 5] がいくつか知られるようになったという事情も考えられる。

並列論理型言語の場合、並列性の記述機能を導入したことによって本来の論理型言語が備えていた理論的明瞭さというが長所失われているのではという懸念が無かったわけではない。しかしながら、CCS、CSP のような「理論的に明解」であることが主眼であるような並列計算モデルに比べて、見かけ上さほどキタナいものでもないという印象 (実際 Prolog のカット (cut) と GHC のコミットを比べれば、go to と if then ぐらいの印象の違いはある。) が、並列論理型言語の形式的意味論の研究への取り組みを勇気付けたともいえよう。

本稿ではこのようなコミット選択型並列論理型言語と、その周辺の言語の形式的意味論研究の流れについて解説する。本稿では論理型言語とその基礎理論に対する初等的な知識[3]及び、並列論理型言語についての基礎的概念の知識を仮定している。[4]

本稿の構成は次のようになっている。まず、2 章で純 Prolog の形式的意味論を拡張して並列論理型言語の同期 / 通信機構のセマンティクスを記述する試みについて単純なアイデアから出発し、その問題点とそれらを解決した結果について解説する。第 3 章では並列計算モデルのセマンティクスに対するアプローチを採用して得られた結果について紹介し、4 章ではこれらの理論の結果をふまえた上で新しい並列論理型言語の拡張について最近の話題を紹介する。

なお本稿では GHC、PARLOG、Concurrent Prolog 等の言語を扱うが、特に断りの

[3] [47] を参照されたい。

[4] [72, 73, 24] を参照されたい。言語族全般については [68] が優れたサーベイである。

ない限りそれらはFlat Guardに限定したサブセットを意味する。また説明を簡単にするため、システムに組み込みの述語としては単一化のみを許すことにする。

2 古典的 / 宣言的アプローチ: 論理型言語の意味論として

2.1 方向性のある入出力関係

Horn論理を用いた純Prologのような論理型言語においては、プログラムのセマンティクスを最小モデルによって形式的に定める方法が提案されている[1, 47]。

このアプローチによれば純Prologのプログラムは、SLD導出によって定まる成功集合すなわち操作的セマンティクスと等価な宣言的セマンティクスすなわち最小モデルセマンティクスを、導出戦略のような操作的概念を用いることなく独立に定めることができる。さらにこの最小モデルセマンティクスは、プログラムの構文(すなわち記号列)からセマンティクス領域へのある写像を適用することによって定まる意味論、すなわち不動点的セマンティクスと等価なものであることが示される。プログラムのセマンティクスが構文領域から意味領域への写像によって定まるという性質は、表示的意味論が徹底してこの立場を追求していることからわかるように、プログラムのセマンティクスの理論的有用さを決定づける重要な性質である。

このように見通しのよい形でセマンティクスが定まることが論理型言語の長所のひとつであるとうたわれてきた。

このような論理型プログラムのセマンティクスに対する古典的なアプローチでは、直観的には節の(有限)集合であらわされたプログラムのセマンティクスを、ゴールが成功した時に最終的に出入りした代人の総和に着目したとき、もとのプログラムと等価となるような基底単位節の集合によって記述している。このような基底単位節の集合の各要素は、もとのプログラムの具体的な個々の計算(のあるパターン)に入出力代入の合成をほどこした結果となっている。純Prologのプログラムの場合、各ゴールのどの引数が入力 / 出力となっているかを区別しないで、すべての方向の計算をひとつの基底単位節で表わしている。

並列論理型言語のプログラムは(PARLOG, Concurrent Prolog, GHCいずれの場合も、そのpureなサブセットでは)、コミット記号を論理積と解釈しそれ以外の同期 / 通信のための記号を無視すれば、通常の純Prologと同じように読むことができ、純Prologの場合と同様な方法によってその最小モデルを得ることができる。しかしながらこのようにして得られた最小モデルはプログラムのセマンティクスにはなり得ない。一般にはその計算規則によって得られる結果以外のものを含みうるからである。(言い換えれば、操作的

意味論が完全性を持たなくなる。)次の例を考えよう。 ここではGHCでプログラムで記述している。

(例1) [70]

```
p(X, Y) :- X = [A,B|X1] | Y = [A, B]

merge([A|X], Y, Z) :- true | Z = [A|Z1], merge(X,Y,Z1)
merge(X, [B|Y], Z) :- true | Z = [B|Z1], merge(X,Y,Z1)
merge([] Y, Z) :- true | Z = Y
merge(X, Y, Z) :- true | Z = X

dup([A|I], O) :- true | O = [A, A]

s(Ix, Iy, Out) :- true | dup(Ix, Ox), dup(Iy, Oy),
                  merge(Ox,Oy,Oz), p(Oz, Out)

plus1([A|In], O) :- true | A1 = s(A), O = [A1]

t(In, Out) :- true | s(In, Mid, Out), plus1(Out, Mid)
```

このプログラムはBrock-Ackermannの例題[4]として知られているものの一部である。ここでは詳しい説明は省略するが、コミット記号を論理積と解釈してプログラムの最小モデルを考えると、t([0], [0, s(0)])はその元となることは容易にわかる。実際にこの基底アトムはこのようなプログラムのもとで計算すると成功するゴールである。しかしながら、次のような第2引数が変数であるようなゴールを考える。

```
?- t([0], Out).
```

このとき、実際はt([0], [0, s(0)])は結果として計算されることはない。それはコミット機構の制御によるものである。(例1終わり)

並列論理型言語のプログラムの場合、各ゴールが計算の途中で扱うデータは、その各部分が制御機構によって「能動的に生産できる」部分と「受動的に与えられ消費されるべき」部分に分けられている。言い換えれば、プログラムは暗に(又は陽に)入出力の方向を指定している。 その結果先の例に見られたように、各ゴールにとってある入出力方向の計算は可能であるが、別の方向の計算は不可能なものとなりうる。

ここではまず、並列論理型言語のセマンティクスの古典的アプローチとして、各ゴールの各引数を具体化している各関数記号について、受け取ったもの / 自ら具体化したものの区別を陽に表記することによって、基底単位節の表現する計算に方向性を導入した方法

[44] について紹介する。 この方法は基底単位節にあたるものを、ボディ部分に項の単一化以外のサブゴールの呼び出しが出現しない GHC の節、すなわち GHC の単位節で表現する方法である。

Var を可算無限個の変数の集合, *Fun* を関数記号の集合とする。*Fun* の各要素に対しては arity がそれぞれ定まっているものとする。*Var* と *Fun* の元から通常のように定まる項の集合を *Terms* であらわす。ここで *Terms* の上の前順序関係を $t_1 \preceq t_2$ iff $\exists\theta : t_1\theta = t_2$ と定義する。ここで θ は代入 (すなわち、変数から項への写像) である。$\preceq$ の対称閉包は**呼び換え** (variance) と呼ばれる同値関係: $\sim$ を定める。*Terms* の呼び換えによる商集合 *Terms*/ $\sim$ を *TERMS* と表記する。これが通常の論理型プログラムのセマンティクスにおける Herbrand 領域にあたる。通常の Herbrand 領域の場合、変数を含まない基底項のみからなる領域を考えるが、並列論理型言語の場合は変数を含む項、あるいは変数を含む項を結果として許す代入をセマンティクスの領域に含める場合が多い。本稿で紹介する方法も代入を用いるアプローチにおいては、変数を含む項を結果として許している。このほうが自然な記述を可能にするからである。その理由について述べる。

通常の Prolog の場合、与えられたゴールが計算終了した段階でゴールに出現した変数が具体化されないで変数のまま残されていた場合、その変数は全称束縛されたものと考え、任意の項を代入したものが計算結果として考えられる。これは Prolog が Horn 論理から受けついだ性質 (完全性 / 健全性) との整合性上、実際の計算処理との対応には目をつぶっていることによる。しかし、これが不都合を生じないのは Prolog のプログラミングが、少なくとも純 Prolog の範囲では、「変数が具体化されていないこと」を積極的に結果として観測することはありえないことによる。

しかしながら、並列論理型言語の場合変数を含む項が結果に出現することは、その言語の性質上又並列論理型言語によって記述されるプログラムの性格上必然である。次のような例を考えよう。

(例 2)

```
t(Z) :- true | r(X, Y), p(X, Y, Z)
r(X, Y) :- true | X = Y
p(X, Y, Z) : - X = Y | q(X, Z)
q(X, Z) :- X = f(X1) | Z = 0
q(X, Z) :- true | Z = 1
```

この例では、操作的な計算規則に従えば t(Z) というゴールに対して、次のような動作

をする。t(Z) は最初の節にコミットし、サブゴールとしてr(X, Y) とp(X, Y, Z) を呼び出す。r(X, Y) は2行目の節にコミットして、X, Yを同じ変数に具体化するが、いかなる関数記号を含む項も束縛されることはない。

一方、p(X, Y, Z) はX , Yともに変数のままであっても、それが同じ変数であることを観測して3行目の節にコミットすることができる。[5]その 結果、q(X, Z) というサブゴールが呼び出されるが、ここでXは変数項のままである。したがって、4行目の節にはコミットすることができず、5行目の節のみコミット可能である。

すなわち、t(z) というゴールに対してz = 1という結果のみ可能性がある。

r(X, Y) のセマンティクスは基底単位節のみで表現した場合、XとYが同じ基底項に具体化されたすべての基底アトムの集合となる。このことはp(X, Y, Z) がX = Yというガードを解き、q(X, Z) というサブゴールを呼び出すという動作を説明するには支障はない。

しかしながら、ここでq(X, Z) のXに任意の基底項が束縛されていると考えると、q(X,Z) が4行目の節でサスペンドし5行目の節にのみコミットできるという現象が説明できない。

すなわち、 並列論理型言語の場合、ガードゴールが「ある変数が具体化されていることを観測してない」という情報を積極的に用いてサスペンドさせている。
(例2終り)

この例より領域に変数を含めた項を用いることは自然な記述を可能にするものであることが理解できるであろう。しかも全ての変数を同一視するのではなく、変数には区別がつくものとするのが自然である。(さもなくば先の例で、X = Yというガードが機能することが説明しにくい。)

しかしながら、変数名の呼び換えによって同じになる計算を区別する必要はない。そこで多くの場合、変数を含めた項の領域の呼び換えによる同値類 / ～ をHerbrand 領域にあたる領域として扱っている。またこの呼び換えによる同値類 / ～ は、項の集合だけでなく、節、ゴール、代入のそれぞれの集合に自然に拡張して用いる。

述語記号の集合を $Pred$ で表わす。次のようなGHC の単位節を考える。

$p(X_1, ..., X_n) : -I|O$

ここで $p \in Pred$ は arity n の述語記号、$X_1, ..., X_n$ は互いに異なる変数 ($\in Var$)、I, O は それぞれ $v = t (v \in Var, t \in Terms)$ という形の単一化ゴールの並びである。

ガード付アトム (guarded atom) とはこのような単位節のうち、強標準形と呼ばれる条

[5]GHC を基本にした言語 KL1 の一部の処理系では、このようなコミット機能はサポートされていない場合もある。

件を充たすものである。形式的な定義はここでは述べないが、直観的には:

$p(X) : -X = 1 | X = 0$
$p(X) : -X = 1 | Z = 1$
$p(X) : -X = 1 | X = 1$

のような無意味な節や冗長な単一化ゴールを含む節を除いた、意味ある計算を表現するものを意味する。

解釈基底 (interpretation base) とはガード付アトムの集合で $\sim$ による同値類である。これは通常の Prolog の意味論における Harbrand 基底にあたるものである。

このように考えると先の例 (1) では、成功するゴールと成功しないゴールはガード付アトムによって次のように区別される。

$t(I, O) : -I = [0], O = [0, s(0)] \mid true$
は成功する計算を表現するガード付アトムであり、
$t(I, O) : -I = [0] \mid O = [0, s(0)]$
は成功しない。

以上のアイデアは並列論理型言語としては比較的単純な部類である Flat GHC についてのものであるが、Parlog, Concurrent Prolog についても各関数記号に $+, -$ といったモード表記を与えるアプローチ等が提案されてきた [42, 43]。すなわち、$p(X) : -X = f(Y) | Y = 0$ というガード付アトムはこの方法では $p(f^-(0^+))$ のように表現される。[42, 43] では、これによって PARLOG のモード宣言による方向付け又は Concurrent Prolog の読みだし専用表記による方向付け等も定式化を試みている。

これらの方法の表現は、標準的な Herbrand 基底による表記が $p(X_1, ..., X_n)\theta$ という形であるのに対して、

いずれも θ の部分を入力側 θ_i と出力側 $\theta_o (\theta_i \circ \theta_o = \theta)$ (ここで $\circ$ は代入の合成を表わす) に分けて記述することによって情報量を増やしているものと考えられる。いわばこれらの方法が入出力の対の集合 (すなわち入出力関係) による記述であるのに対し、標準的な Herbrand 基底による表記は入出力対の合成結果の集合による表記となっていたといえる。

2.2 入出力履歴による記述

さてこのように「能動的に作る部分」/「受動的に受けとった部分」を区別した表現を用いて新たな解釈基底を定めたわけであるが、これで純 Prolog の宣言的意味論と同様に見通しのよい意味論を与えることができるであろうか?

本節では、この点について問題点を具体的な例を交えて述べ、さらに並列論理型言語の意味論を古典的アプローチの拡張で与える際の解決すべき問題点をあげ、それらを解決している結果として [51] が GHC に対して適用した方法について述べる。

2.2.1 入出力関係による記述の問題点

(1) **合成可能性:**

先にも述べたようにプログラムのセマンティクスが見通しよく定まっているという基準のひとつに、与えられたプログラムについて操作的な定義によって与えられる意味論に加えて、プログラムの構文から意味の領域への写像によってセマンティクスが定まり、両者の等価性が示されなければならないという点がある。

通常の論理型プログラムの場合、与えられたプログラム p から Herbrand 解釈の領域の上の連続関数 T_p を得ることにより、プログラムのセマンティクスは T_p の最小不動点によって定義された [47]。

同様な手法で、与えられた並列論理型言語のプログラムについてセマンティクスを最小モデルという形で定め、さらにプログラムのシンタックスから解釈領域のべき集合の領域の上の連続関数を定め、最小モデルの不動点による特徴付けが試みられた。しかしながら、このことは容易ではない。

ここで、純 Prolog の場合の最小不動点によって意味論を定めていたステップを思い出してみよう。[1, 47] 直観的には、不動点意味論を定める連続関数 T_p は、n-1 ステップ以内で既に成功することがわかっているアトムの集合から、n ステップ以内で成功するアトムの集合を帰納的に求める関数になっている。この関数は、ある節が存在してボディ部分のインスタンスであるサブゴール (節) が成功すればヘッドのインタンスが成功するという原理から成立している。ボディ部分のサブゴール (節) が成功するためには、サブゴールに含まれる各アトムがすべて成功すればよい、すなわちすべてのアトムが既に n-1 ステップ以下で成功するアトムの集合として計算された集合に含まれていればよい。

すなわち純 Prolog の場合このような連続関数が有効である理由として、ゴール節が成功するか否かが、成功するアトムの集合から容易にわかるという点がある。言い換えれば成功するアトムの計算を表現するものから成功するゴール節の計算を表現するものが得られなければならない。

しかしながら、並列論理型言語の場合はこの点が自明ではない。すなわち、成功するガード付アトムの集合から、成功するゴール節の集合を得る方法は自明ではない。それどころか方向性を導入した入出力関係だけでは不可能であることが示されるのである。このことは、入出力関係だけを記述する解釈基底では、並列論理型言語の見通しよい意味論を

与えるのには不十分であることを示している。

まず、入出力関係による表現が「成功するゴール節」を特徴付けるのに不十分であることを示す有名な例題を紹介する。前節で紹介した(例1)はその一部である。ここではその全体を紹介する。この例題はBrock-Ackermannの例題と呼ばれ、最初[4]によってデータフロー型の言語について、「非決定性とフィードバックを含むデータフローグラフ」の意味論を与える場合において、[38]等で提案された入出力関係によるアプローチが不十分であることを指摘する例として用いられた。その後、[70]でGHCで記述した同様な例が紹介された。

(例3)[4, 70]

先の(例1)のプログラムを考えよう。ここで、p(X, Y)の定義を次のような節でおきかえてみる。

```
p(X, Y) :- X = [A|X1] | Y = [A|Y1], p(X1, Y1)
p1(X1, Y1) :- X1 = [B|X2] | Y2 = [B]
```

置き換えられた節はp(X, Y)というゴールに注目する限り、先の定義と同じガード付アトムの集合を成功集合として持つことは容易に理解できよう。すなわち、いずれも第1引数に与えられたリストの要素のうち最初のふたつを切り出して、第2引数に出力するプログラムになっている。

ここで、このプログラムで成功するガード付アトムの集合のうち、t(In, Out)という形のヘッドを持つものを考えよう。先に述べたように(例1)では、

```
t(In ,Out) :- In = [0] | Out = [0, s(0)]
```

という結果は計算されない。しかしながら、p(X, Y)の定義を置き換えたプログラムでは、この結果は可能である。(例3 終り)

もしこのようなプログラムのセマンティクスが解釈基底のべき領域の上の連続関数によって得られるとし、その関数が純Prologの場合と同様にボディ部分のインスタンスであるサブゴールの計算を参照してヘッド部分のインスタンスのゴールの計算の集合の要素を構成するような形をしているとしたら、このようにサブゴール部分の定義を置き換えても、置き換えた部分の計算を表現する入出力代人の関係が等価なものである以上、全体の結果に影響をおよぼすことは無い筈である。 ゆえに入出力関係による記述では純Prologと同様な原理の連続関数を用いてその不動点でセマンティクスを特徴付けたのでは、t(In, Out)のような例で、一部を置き換えたときの影響が表現できない、すなわち入出力関係によって各アトムのセマンティクスを記述するだけでは、ボディ部分で呼び出され

るサブゴール節の計算を表現する入出力関係を構成できないのである。

この問題は一般的にはセマンティクスの合成可能性 (compositionality) の問題である。プログラムは通常の感覚では、いくつかの部品を結合することにより成りたっていると考えるのは自然であろう。並列プログラムの場合、もっとも典型的には一つ一つのプロセスが部品であり、並列合成 (parallel composition) を結合演算としてそれぞれの部品が組み立てられることによって実際に動作するプログラムができあがる。このような場合、暗黙のうちに我々はプログラム全体の機能が、各部品の機能から完全に予測されることを前提としている。

この性質は、次のような一般的な形で述べることができる。あるプログラム (構文的な記号列としての) の断片 p_1, p_2 が与えられたとき、ある構文上の合成演算 $*$ によってプログラム $p_1 * p_2$ が構成される。ここで、$p_1 * p_2$ の意味は p_1, p_2 それぞれの断片の意味 s_1, s_2 からプログラムの意味を構成する演算 $*_{sem}$ によって $s_1 *_{sem} s_2$ として与えられる、言い換えれば構文領域から意味領域への関数をセマンティクスを与える関数を sem とするとき、

$$sem(p_1 * p_2) = sem(p_1) *_{sem} sem(p_2)$$

が成立するようなとき、このセマンティクスは合成的 (compositional) に定まるという。一般にこの性質はプログラムの変換、検証等のセマンティクスの応用を考える上で極めて重要なものである。特に純 Prolog と同様な原理を用いた連続関数の不動点でセマンティクスを定義したければ、並列合成についてのこの性質は必須なものとなる。

(2) 作用 *cdot* 反作用的動作の記述

先の例は入出力関係のみでは不動点意味論を定義することが基本的には不可能であることを示すものであるが、実は入出力関係のみでは不都合が起こることはこれだけに留まらない。並列論理型言語では、並列性を陽に意識したプログラミングに固有のパラダイムに対応するために、いくつかの拡張を求められることになる。

古典的アプローチはもともとの純 Prolog のような論理型言語意味論として提案されたものであり、その探索 / 検索といった Prolog のスタイルに依存した意味論の形といえる。すなわち純粋な論理型プログラムの世界では、計算とはゴールの反駁であり、最初に質問がゴールの形で与えられ、有限な計算の後に答の代入が得られるという計算を想定していた。 それゆえに形式的意味論で用いられる「計算の表現」はゴールの起動時と停止時に行なわれる入出力の対を記述したものとなっていた。

一方、並列論理型言語の世界では、GHC (KL1) の PIMOS、PARLOG の PPS、Concurrent Prolog の LOGICS などのようなオペレーティング・システム的な応用が当初か

ら意識され、このようなオープンなシステムの記述言語としての期待があった。これらの応用においてはシステムと外部の環境あるいはシステム内部のプロセスの間で対話的な通信を行う形の計算が行われる。このような計算を作用 *cdot* 反作用的 (reactive) な動作と呼ぶ。並列論理型プログラムにおける並列プロセスでは対話の単位となるのは Prolog のようにゴールではなく、ストリームとして用いられる項 (多くの場合はリスト) の要素の具体化である [72]。

このような事情より、並列論理型言語のゴールは完結した形で与えられて完結した答えを待つキュエリではなく、ストリームによって外界と結ばれ常に入出力を続けながら計算を続行する無限プロセスであるという見方が日常的なものとなっている。 すなわち並列論理型言語で記述されたプログラムにおいては、「計算を外界との通信の過程としてとらえる」 [73] 見方が自然なものとなる。一般にプログラムのあるプロセス p(サブプロセスを含む) の動作は、並列に走る他のプロセス等の観測者からみた p が行なう入出力であると考えられる。並列論理型のプログラムの場合、観測者は p の起動と停止の時にのみ通信を行なうわけではなく、変数の具体化情報を実行途中にやりとりすることができることになっている。 このことより並列論理型言語の形式的意味論においては、ゴールの「計算」を表現するものも、このような作用 / 反作用的な「通信の過程」を表現する数学的実体であることが望ましい。 すなわち、このような並行計算系セマンティクスが定めるプロセスの上に同値関係は、通信の過程を観測しているものでなければならない。

入出力関係による計算の表現はこのような立場からは、起動時と停止時にのみ通信を行なう特別な場合の計算の表現となっており、作用 *cdot* 反作用的動作の記述としては不十分なものとなる。

作用 *cdot* 反作用的動作を記述するモデルとしては、CCS [49], CSP [35] 等の並列プロセスのモデルがあげられるが、これらのモデルにおける手法を論理型に適用したセマンティクスについては次章に譲ることにして、ここでは無限プロセスを含むプログラムの作用 *cdot* 反作用的で合成的なセマンティクスを宣言的意味論の形で与えた例 [51, 53] について解説する。この方法は [44] の方法を拡張して通信の過程を表現するような要素の集合で Flat GHC のセマンティクスを定義する方法である。

2.2.2 ガード付ストリーム

このアプローチでは、[44] における ガード付アトムの概念の拡張にあたる概念を用いる。ガード付アトムが入出力関係の各要素の表現であったのに対し、入出力履歴は通信の過程を半順序集合で記述したものである。

入出力履歴はプロセスの呼ばれたときの形を表わすヘッド h とプロセスのある実行における入出力を示す gu を用いて次のように表わされる。

$$h :- gu$$

ここで h は互いに異なる変数に述語記号を適用したもの，gu は**ガード付きストリーム**と呼ばれるもので、ガード付単一化と呼ばれる対の集合のうち GHC のプログラムの計算として意味をなすようなものである。**ガード付単一化** (guarded unification) とは、ある節にコミットするまでに通過するガードを解くために必要な入力代入 σ とコミットした節のボディ部で実行される単一化ゴール u_b の対 $<\sigma|u_b>$ である。直観的には ゴールの引数が σ によって具体化されると u_b という単一化が行われることを意味する。

与えられたガード付単一化の集合 gu の上に次のような関係 $\prec$ を定義する。$<\sigma_1|u_1>, <\sigma_2|u_2> \in gu$ とする。$\sigma_1 \subset \sigma_2$ かつ $\sigma_1 \neq \sigma_2$ のとき、

$$<\sigma_1|u_1> \prec <\sigma_2|u_2>$$

とする。このように定義された関係 $\prec$ が呼び換えによる同値類を考えれば、半整列順序になることは容易に示せる。

直観的には、$<\sigma_1|u_1> \prec <\sigma_2|u_2>$ とは、u_1 よりも u_2 の方が、実行するまでにより多くの具体化を必要とする、ことを意味する。言い換えれば、半順序 $\prec$ は、プログラムのある動作を表現するガード付単一化の集合 gu では、各元の実行順序に対応している。

すなわち、$<\sigma_1|u_{b1}>, <\sigma_2|u_{b2}> \in gu$ について、$\sigma_1 \prec \sigma_2$ ならば、u_{b1} は u_{b2} より先に実行可能となる。

ガード付単一化の任意の集合が GHC の計算として意味をもつわけではない。

(例 4)

ガード付単一化の集合 gu を次のとうりとする。$gu = \{<\{X = [0|X1]\}|Z = [0|Z1]>, <\{X = [0|X1], X1 = [1|X2]\}|Z1 = [1|Z2]>, <\{X = [0|X1], X1 = [1|X2], Y = [0|Y1]\}|Z2 = [0|Z3]>, ...\}$

このとき、

$\texttt{merge}(X, Y, Z) :- gu$

は (例 1) で定義されたプログラム merge の可能な計算を表現している。すなわち、X, Y という二つの入力ストリームに、{ X = [0,1,...], Y = [0,...] } という具体化が与えられたとき、最初に X からの入力を先に 2 つ処理し、続いて Y からの入力を処理した場合の

計算を表現している。この場合、出力 ストリーム Z に現れる出力結果 (の途中経過) は Z = [0,1,0,...] となる。

一方、$gu = \{< \{X = [0|X1]\}|Z = [0|Z1] >, < \{X = [0|X1], Y = [0|Y1]\}|Z1 = [0|Z2] >, < \{X = [0|X1], X1 = [1|X2], Y = [0|Y1]\}|Z2 = [1|Z3] >\}$ とすると、

$\mathtt{merge}(X, Y, Z) :- gu$

は同じ入力に対して、別の順序で処理をした、(言い換えれば、コミットする節の選択が異なる) 計算を表現しており、この場合は出力 ストリーム Z に現れる出力結果 (の途中経過) は Z = [0,0,1,...] となる。

しかしながら、次のようなガード付単一化の集合 gu' はいかなる正常な計算も表現しない。

$gu' = \{< \{X = [0|X1]\}|Z = [0|Z1] >, < \{X = [0|X1], Y = [0|Y1]\}|Z1 = [0|Z2] >, < \{X = [0|X1], X1 = [1|X2], \}|Z1 = [1|Z2] >\}$

何故ならば、Z1 について第 1 番目の要素が 2 度異なる値に具体化されているからである。

(例 4 終わり)

この例にあらわれる gu のように、ガード付単一化の集合が Flat GHC の計算を表現するとき、**ガード付ストリーム** (guarded stream) と呼ぶ。与えられたガード付単一化の集合がガード付ストリームであるか否かは、いくつかの条件により定まる。[53] その条件とはここでは述べないが、先に述べたガード付アトムの標準形の条件と基本的な考えかたは同じである。

入出力履歴は先に述べたガード付アトムに比べて情報量が増えている。すなわち、計算の結果得られた代入に対して入力 / 出力の区別をつけただけでなく、出力された部分について、その部分を具体化するのに必要とした入力代入を出力の各要素毎に対にしている。すなわち入出力の因果関係を記述したものになっており、その結果入出力履歴はゴールの作用 *cdot* 反作用的動作の記述となっている。

[53] の方法は、このような入出力履歴の集合を解釈基底とし、プログラムのセマンティクスは可能な計算を表現する入出力履歴の集合で与える方法である。このような方法を用いれば、先の (例 3) に出現した p(X, Y) について、置き換える前と後のプログラムは次のような入出力履歴を含む / 含まないプログラムとして区別することができる。

$p(X, Y) : -\{< \{X = [A|X1]\}|Y = [A|Y1] >, < \{X = [A|X1], X1 = [B|X2]\}|Y1 = [B] >\}$

このようにプログラムのセマンティクスを入出力履歴で与えるように定義した目的は、純 Prolog と同様に「ゴール節の動作」を「サブゴール (アトム) の動作」から再現できる

ようにすることであった。実際にこのアプローチでは同期付マージと呼ばれるガード付ストリームの結合演算を導入している。この演算は、プロセス $g_1, g_2, \ldots, g_n$ が並列に走った際の計算を表現するガード付ストリームを、各 g_i の計算を表現するガード付ストリーム gu_i から合成するものである。形式的な定義は [53] を参照されたい。基本的なアイデアは以下の通りである。$gu_1, \ldots, gu_n$ の同期付マージの結果 ($gu_1 \| \cdots \| gu_n$ と表記される。) は $gu_1, \ldots, gu_n$ のすべての和集合の各元を次のような規則で置き換えたものである。

$< \sigma_i | X = \tau > \in gu_i$ かつ $< \sigma_j | u > \in gu_j$ で $(X = \tau) \in \sigma_j$ であるものとする。すなわち i 番目のプロセスが X のプロデューサで j のプロセスがコンシューマであるとする。このようなとき、$< \sigma_j | u >$ の替わりに $< \sigma_i \cup \sigma_j \setminus \{X = \tau\} | u >$ を $gu_1 \| \cdots \| gu_n$ の元とする。

直観的には、あるプロセスが u を実行するために X にある入力 τ を待っているとき、τ のプロデューサが並列に走れば、τ に替わってプロデューサが X を τ に具体化するのに必要となる入力を待って u が実行される様子を表わしている。先に述べた Brock-Ackermann と同様な例について、t(In, Out) というゴールの可能な計算のみが各サブゴールの計算から同期付きマージを用いて合成される様子が [76] に示されている。

ガード付ストリームが代入の拡張概念であると考えるならば、同期付マージは代入の結合 (combination) の拡張概念となる。

この同期付マージ演算を用いることにより、与えられた GHC のプログラムに対して、純 Prolog の場合の T_P に相当するような解釈 (すなわち入出力履歴の集合) の領域の上の連続関数を定めることが可能となる。その結果与えられた GHC のプログラム P について、P の構文からそのセマンティクスを構成することが可能となった。

2.3 古典的アプローチのもうひとつ課題

前節では入出力履歴によってプログラムの動作を表現することによって論理型プログラムにおける古典的なアプローチを GHC に拡張した。ここで入出力履歴によって表現されていた動作はプログラムの正常な動作のみであった。すなわち古典的アプローチでは基本的には正常な動作のみを扱い、失敗 / デッドロックする動作のことはとりあえず考慮されていなかった。

純 Prolog の場合は定理証明の過程を計算と解釈するものであり、そのために実行はめざす結果を探索して可能な実行を捜しまわるものを想定している、すなわち don'nt know な非決定性を想定した言語といえた。すなわち最小モデルがすべての可能は計算結果の集合を表現しているとは、適切な選択を行なうことによって解にたどりつくことが可能であることを示しており、結果として関心があるのは「その結果に至る反駁が存在するか否

か」であり、実際の(逐次型の)Prologはこの探索の機能をバックトラック機能によって実現している。

一方、並列論理型言語の言語ではコミット機構を導入して、一度選択した節は二度とキャンセルすることはできないような仕様となっている。これらは主に並列分散環境での効率的実現を考慮していたという事情による。すなわち、言語が当初提案された背景から、don't careな非決定性言語とならざるを得ないのである。

このようなdon't careな非決定性言語においては、ある節の選択の結果実行が失敗する可能性の有無は本質的にプログラムのセマンティクスに影響する。すなわち、純Prologのようなdon't knowな非決定性を扱っている場合には成功集合のみでプログラムのセマンティクスを記述すれば、(プログラムが暴走する場合は別として)プログラム検証の立場における全面的正当性を議論するのに充分であったが、並列論理型言語の場合には、成功集合のみでは部分的正当性のみを記述していることとなり、「失敗の可能性のあるゴール」の集合をも同時に記述しなければ全面的な正当性を議論することはできない。

そのために古典的アプローチの手法を用いて失敗する動作の定式化もいくつか報告されている[21]。

3 プロセス指向の意味論

前節ではvan Emden, Lloydらの古典的アプローチを拡張し、GHCプログラムの直観的な操作的意味論の定める計算結果と一致する最小モデルの定義を与えることを試みた。古典的アプローチでは、ゴールの形(パターン)とそのパターンの計算を表現する具体化(それは代入の拡張概念)の対を元とする解釈基底のべき領域を意味領域としていた。一方、当初から並列論理型言語は構文的には純Prologの拡張ではあるものの、前節で述べたようにそのプログラミング作法(パラダイム)からProlog等とは異なるパラダイムの、CSP, CCS, データフローモデルあるいはリダクション モデルに近い言語と考えることができる。

それゆえそれらの並列計算モデルの理論についての結果を並列論理型言語の意味論に適用したアプローチが、いくつか報告されている。実際、前節で述べた並列言語のパラダイムから来る形式的意味論への要求はTheoretical CSP [5], シナリオ モデル[4]、Plotkinらの状態遷移モデル等の並列計算モデルの形式的意味論を提案する際に問題にされたことである。したがってこれらの問題の一部については古典的なアプローチを拡張して解決を試みるより、Horn論理に基づく構文の論理型独特のスタイルに合わせて変形するという課題を解決できれば、既に存在する並列計算モデルの理論の結果を適用した方が有利であ

る場合も当然考えられる。実際、前節の述べた合成的な意味論の問題については [57] でシナリオ モデルと呼ばれるデータフロー モデルの意味論で用いられてきたモデルを Concurrent Prolog のサブセットに適用した結果によって比較的早くから試みられているのである。

この章で話題とするのは合成的に定義できるセマンティクスを与える上で同時に派生したもうひとつの問題であり、プロセス的意味論によって解決が得られた意味論の**充分な抽象性** (fullly abstractness) の問題である。

前節で述べた例では、プログラムのセマンティクスを入出力関係で記述しただけでは合成的にセマンティクスを与えることは不可能であったが、計算途中の情報をもつけ加えた入出力履歴で記述することによって可能となった。このことからわかるように、セマンティクスが合成的に定義されるためには抽象化に限界があり、ある程度の情報量が各プログラムの表示に含まれていなければならない。このことから最も自明に合成的な意味論を構成する方法は、構文領域それ自身を意味領域としてしまうことであろう。しかしながら、明らかにこの方法は何のメリットもたらさない。もしこのアプローチを採用するならば、そのセマンティクスのもとで等価性を保存する有用なプログラム変換を考えることは殆ど不可能となる。したがって意味あるセマンティクスとは合成的に定義され、(1) 通信の過程を観測することによって区別可能なプロセスを異なる同値類に類別し (2) できるだけ分割の荒らい同値類を定めるものであることが好ましいことがわかる。

意味論の抽象性についての議論は PARLOG, GHC , Concurrent Prolog 等の言語に依存することなく、ほとんど共通に議論されてきており、あるいはデータフローネットワーク、等の他の並列計算モデルの理論の枠組みを論理型に適用するような形で解決されてきているものも多い。本節ではこのような並列プロセスについての意味論を並列論理型言語に適用して充分に抽象的な意味論を与えた試みの例として Gerth Codish Lichtenstein Shapiro の意味論 [15] について紹介する。

3.1 Gerth Codish Lichtenstein Shapiro の意味論

Gerth らの結果 [15] は (Flat) Concurrent Prolog のサブセット TFCP(Theoretical FCP) に対して報告されたもので、Plotkin 風の状態遷移モデルが基本とする操作的意味論と、それと等価で充分に抽象的な表示的意味論を示している。この結果では操作的意味論 / 表示的意味論共に、並列動作を逐次的な非決定性に帰着した形で記述している。

操作的意味論の概略は以下の通りである。代入の集合を B で表記する。状態遷移システムは次のように定義される状態の集合 Q と、遷移関係 $\rightarrow \subset Q \times B \times Q$ からなる。状態とは二項組 $< a;\theta >$ である。ここで a はゴール節又は、成功、失敗、デッドロックを表

わす記号 (tt, ff, dl) のいずれか、θ はその時点で (受動的あるいは能動的に) 得られた代入である。

以上からわかるように、この操作的意味論では計算の実行途中の代入及び計算されているゴールの形が観測可能なものとなっている。

一方表示的意味論の方については、Theoretical CSP で提案された**失敗例集合意味論** (failure set semantics)[6] [5] を基本にしている。CSP の失敗例集合意味論では、プロセスのセマンティクスは失敗例 (failure) の集合である。プロセス P の失敗例とは、P が実行した事象の系列 s と、s の実行後ある集合 X のいかなる元を環境が P に期待しても、もはや P は先へ進むことができないような集合 X の対である。直観的には P は s を実行しその先は X に含まれるいかなる動作も実行できないことを意味する。このような失敗例集合によるセマンティクスは、CCS ではデッドロックの有無を区別する最も粗い同値関係を定義するものとして知られている [35]。

ここでは Concurrent Prolog の計算において CSP の事象に対応する概念として、代入の読み込み (input) と書き込み (output) を考える。代入 θ の読み込み事象を θ^I、書き込み事象を θ^O で表記する。計算の履歴はこのような書き込み事象と読み出し事象の集合を並べた系列によって表現される。ここで計算の履歴の領域 SEQ の元となるのは、読み込み / 書き込みによってやりとりされた束縛情報が矛盾のないものとなるときのみである。この条件は詳しくは述べないが、前節で触れたガード付きストリームの条件と本質的に同様なものである。ここではプログラムの表示 (denotation) は**中断例** (suspension) (c, s) の集合 F である。各 (c, s) は計算の履歴を表現する系列 $c \in SEQ$ と集合 s からなる。ここで s はこれ以上出力を出さなくするような入力代入であるか、tt(成功) 又は ff(失敗) という停止記号か、不完全な計算を表現する空集合記号 $\emptyset$ である。直観的にはプログラム P の表示に (c, s) が含まれるとは、P には c という計算が存在し、その結果 s という代入を読みとることができない、あるいは s という代入と矛盾する状態に至ることを表わしている。中断例の領域を $SUSP$ と表記する。

勿論 $SUSP$ の任意の部分集合がプログラムの表示になりうるわけではない。ここでいくつかの閉包条件を定め、そのような性質を充たすもののみを集めた領域がプログラムの表示の領域となる。すなわちプロセスの表示の領域を DEN とすると、$DEN \subset 2^{SUSP}$ であり、各 $F \in DEN$ は次の閉包条件 (L1) ... (L9) を充たすものである。以下で ε は空

[6] failure set という言葉は従来の論理型プログラムの分野でも用いられる言葉であるが、いくらか意味が異なるので注意が必要である。すなわち失敗する可能性のあるゴールの基底例の集合を、成功集合と対をなす意味で失敗集合 (failure set) と呼ぶことがある。ここではこれと区別するため CSP での用語の failure set を失敗例集合と呼んでおく。

系列、:は系列の連接、$c \preceq c'$ は系列 c が c' のプレフィックスであることを表わす。また $s \subset s'$ は代入 s' が s より具体化していることを表わす。

(L1) $(\epsilon, \emptyset) \in F$

(L2) $(c : c', S) \in F \Rightarrow (c, \emptyset) \in F$

(L3) $(c, s) \in F, s' \subset s \Rightarrow (c, s') \in F$

(L4) $(c, s) \in F, (c, tt) \notin F$ とする。このとき:

$(c : \theta^I, \emptyset) \in F, \forall \theta'(c : \theta^I : \theta'^O, \emptyset) \notin F$ ならば、ある θ'' について $(c, s \circ \theta'') \in F$ ただし θ はどの変数も高々深さ 1 の項にしか具体化せず、$\theta'' \circ \theta'' = \theta$.

(L5) $(c, s) \in F$ ならば任意の (c, s) の呼び換え (c', s') について、$(c', s') \in F$

ここで中断例の呼び換えは代入の呼び換えを用いて自然に定義される。ここで (L1) は、何もしていない計算は、どんなプロセスにとっても計算途中であることを意味する。(L2) は $c : c'$ という履歴を実行して中断に至るならば、c を実行した段階では計算の途中であることを意味する。(L3) は、ある時点で入力代入 s を与えられたとき出力ができなければ、s より具体化の少ない入力を受け取ってもやはり出力ができないことを意味する。(L4) は通信が非同期であることを意味する。(L5) は変数名の呼び換えについて計算を区別しないことを意味する。

以下の 3 つの規則は抽象性のために本質的な役割をはたす。

(L6) $(c : \theta^I : c', \emptyset) \in F$ iff $(c : \rho^I : \theta'^I : c', \emptyset) \in F$

ここで ρ は呼び換えで、$(\rho \circ \theta') = \theta$ かつ ρ は c, c' に出現する変数へは写像しない。

(L7) $(c : \theta^l : \theta^l : c', s) \in F \Rightarrow (c : (\theta \circ \theta')^l : c', s) \in F$ ただし $l \in \{I, O\}$.

(L8) $(c : \theta^O, s) \in F, X\theta \in Var \cup \{V? | V \in Var\}, \neg\exists(c : \theta'^I : \theta''^O : c', s) \in F$

ここで、θ'' は θ に X 以外で等しく $\theta''X = X, X\theta' = X\theta \Rightarrow \exists c'' : \sigma^O \preceq c$ かつ X は σ によって読み出し専用変数として導入される。

(L6) は、ある入力代入を読み込む計算が存在するならば、その入力代入が呼び換えと実際の具体化の 2 つの段階に分かれて実行されうることを示す。また (L7) は入出力いずれの場合も計算の途中で、ある代入 θ の出力 (入力) に続いてさらに代入 θ' の出力 (入力) を観測したら、それらを一度に出力 (入力) したかのように観測される場合も存在することを示す。これらの場合について、一方が存在し一方が存在しない場合と両方存在する場合のプログラムは、観測同値によって区別することができないプログラムであると考えている。(L8) はある計算の途中で導入された変数が消費されないならば、その変数は前もって読み出し専用とされていなければならないことを意味している。

最後の規則 (L9) は非決定性の枝分かれが有限であることを意味している。

(L9) $(c, \emptyset) \in F \Rightarrow \{\theta | (c : \theta^O, \emptyset) \in F\} / \sim$ は有限か又は$F = SUSP$

このように定義された領域 DEN は、$F \sqsubseteq F'$ iff $F' \subseteq F$ とすると $SUSP$ を $\perp$ とする完備な半順序領域となる。

ここで定義した領域がどのように充分な抽象性を保証するかについては、本稿の範囲を超えるのでここでは述べない。

3.2 その他のセマンティクス

前節で紹介したセマンティクスがそうであったように、このようなアプローチにおいては古典的アプローチのように、成功する計算 (正常な計算)/ 失敗 / デッドロックする計算をそれぞれ集合で求めるということはせず、すべての計算の可能性をひとつの集合によって表現するのが普通である。一方、古典的アプローチを用いた場合については、いわゆる成功集合と失敗 / デッドロックする計算の集合の対によってセマンティクスを特徴付けるような形になることは前節の最後に述べた通りである。このような方法の一種として成功集合にあたるような正常な計算の特徴付けについては古典的 / 宣言的アプローチを用い、失敗 / デッドロックする計算については [39] で提案された結果を発展させたメトリックの意味領域を用いて記述するというように、古典的な手法の不足を補うような手法も提案され GHC, PARLOG について報告されている [9, 40]。

また [75] では GHC の展開規則の正当性の議論のためにプロセス指向のセマンティクスを与えている。そこでは直接合成的定義、抽象性などについては深く論じられてはいない。しかしながらそこで与えられたモデルは、見掛け上前節で述べた Gerth らの意味論に結果的に近いものとなっている。そこではプロセスに対して観測 / 通信を行なう動作としてトランザクション (transaction) の概念が用いられているが、このはたらきは前節の (L6),(L7) の規則の役割に通じるものがあるといえる。

4 応用、プログラム変換と新たな言語パラダイム

前節までに GHC, PARLOG, Concurrent Prolog 等の言語族について、その形式的意味論の研究の結果をいくつか紹介してきた。この節では、これらの結果の応用又は反省から並列論理型言語を見なおすことによって提案された新たな言語系について簡単に紹介する。

4.1 展開規則と展開規則意味論

そもそもこのような形式的意味論の研究が進められてきた背景には、意味論を厳密に形式的に定義することによってプログラムの変換 / 合成 / 検証といったソフトウェア開発の手法に対する基礎的研究の分野に応用しようという動機があったのである。実際に積極的に進められた研究のひとつに GHC の展開変換 (unfolding transformation) の研究がある。

展開 / 畳み込み (unfolding/folding) による変換については Hope のような関数型言語を対象に提案され、その後 Prolog に持ち込まれいくつかの興味深い結果を残している。その後 GHC についても同様の研究が試みられ、いくつかの結果が示されている [23, 75]。しかしながら、もともと並列性の記述を意識して設計 / 提案されてきた並列論理型言語に、かならずしもエレガントな展開 / 畳み込みの規則を容易に与えることができるわけではない。実際に並列論理型のプログラムの場合は、Prolog では等価性を保存する変換であった素朴な展開 (naive unfolding) が無制限には許されず、場合によってはプログラムのセマンティクスを変えてしまうのである。例えば、例 1 の述語 $p(X,Y)$ の定義は例 3 の $p(X,Y)$ の定義節を展開したものであるが、両者のセマンティクスが異なることは既に見た通りである。 また仮に何らかの展開規則が与えられたとしても、それらの規則が形式的意味論で定義された等価性を保存することは別に証明すべきこととなる。しかもその証明は自明ではない。

一方ここで考え方を変えて、"展開規則で変換して結果保存されるものがプログラムのセマンティクスである" という考え方がもし可能ならば以後の議論の見通しはずっとよいものになる。しかしながら、このようなセマンティクスが有用なものとなるためには、展開規則によって保存される同値類と、セマンティクスとして有用な概念の定める同値類が互いに整合のとれたものでなければならない。 ここで、次のような結果が知られている。純 Prolog の場合はある性質を充たす展開規則を持つ言語ならば、展開規則によって閉じた同値類のあるものが最小不動点意味論と等価であることが示せる [45]。 すなわち、プログラムのある節から開始して展開規則の適用を繰り返すことによって到達可能な単位節の集合を、そのプログラムのセマンティクスと定める。この定義によって展開規則が等価性を保存することは自明となる。

このような定義が有用なものとなるための展開規則系の性質として、**展開規則の完全性**が提案された。 展開規則の完全性とは、自明でない展開が常に可能であるような性質のことを言う。純 Prolog の場合、このような完全性を持つ展開規則を得ることは困難ではない。

しかしながら GHC にそのままで完全な展開規則を与えることは、既に報告されている

結果 [23, 75] を見るかぎり困難なものであろう。 そこで GHC をもとに完全な展開規則を与えることが可能であるような新たな言語として NGHC (Nested GHC)[22] が提案されている。

NGHC は GHC の拡張であり、多くの点で GHC の特徴を受けついでいる。NGHC で主に新しく導入される概念は多段のガードである。すなわち、NGHC のプログラムの任意の節は m 層のガードをもつ。各ガードは $I|O$ という形の対であり、I,O はそれぞれ入力制約、出力制約である。このふたつはコミット記号 (|) によって区切られている。入力制約 I は**一方向単一化** (one way unification [34]) ゴール $t <= s$ の有限集合である。また出力制約は通常の単一化ゴールの有限集合である。直観的には入力制約は環境に対する条件判定部であり、環境に対して出力をすることはできない。また、出力制約は環境に対して値を出力することができる。

NGHC のプログラムは次のような節の有限集合である。

$$H : -I_1|O_1, I_2|O_2, \ldots, I_n|O_n \rightarrow B_1, \ldots, B_m.(n, m \geq 0)$$

NGHC における単位節とは、$m = 0$ であるような節、すなわち次のような形の節を言う。

$$H : -I_1|O_1, I_2|O_2, \ldots, I_N|O_n.$$

NGHC の計算規則は基本的には FGHC のそれと同様なものと考えられる。すなわち、FGHC におけるサスペンションの規則を一方向単一化によるサスペンションの規則でおきかえたものと考えることができる。与えられたプログラム W のもとでの、$: -B_1, \ldots, B_m.$ というかたちのゴール G の実行とは、G を W に含まれる節を用いて並列にリダクションを行なうものである。計算結果は、FGHC の場合と同様に定義される。NGHC における各リダクションのステップは非決定的な and 並列導出である。

加えて NGHC のプログラムの実行は、次の条件を充たすように進められる。

(1) **ガード部分の逐次的評価**: 各、$I_i, (1 \leq i \leq n)$ は O_i より先に評価される。また、$1 \leq i < j \leq n$ のとき、$I_i|O_i$ は $I_j|O_j$ より先に評価される。

(2) **サスペンション規則**: 各、$I_i = \{t_1 <= s_1, \ldots, t_n <= s_n\}, (1 \leq i \leq n)$ は $(t_1, \ldots, t_n) <= (s_1, \ldots, s_n)$ として実行される。この一方向単一化がサスペンドするとき、このガードはサスペンドする。

(3) **コミット規則**: A と単一化可能なすべての節について、並列に評価を進める。

$C = H : -I_1|O_1, I_2|O_2, \ldots, I_n|O_n \rightarrow B$

を評価中であるとする。今 I_k の評価が成功したものとする。このとき、$k = n$ でかつ他にコミットできる節がなければこの節が選ばれる。$k < n$ であれば、$H : -I_1|O_1, \ldots, I_k|O_k$ という形以外のすべての節が、コミットする候補からはずされる。

[22] で述べられた展開規則は、完全な展開規則となっている。すなわち、任意の強標準形 (これは先に述べた GHC の強標準形の拡張として自然に定義される。) のプログラムの任意の節についてそれが単位節でない限り自明でない展開が可能となる。ここでは展開規則自身については詳しくは述べないが、次の例で説明する。

(例 5) 次のような NGHC のプログラムを考える。

$$\begin{aligned} P = \{\ & t(X,Y) : -\{Y <= b\}|true, \\ & r(X,Y) : -\{X <= a\}|\{Y = b\}, \\ & s(X,Y) : -true|\{X = a\} \rightarrow t(a,Y), \\ & q(X,Y) : -true|true \rightarrow r(X,Y), s(X,Y)\} \end{aligned}$$

展開規則を適用した結果は次のようになる。

$$\begin{aligned} P' = \{ & t(X,Y) : -\{Y <= b\}|true, \\ & r(X,Y) : -\{X <= a\}|\{Y = b\}, \\ & s(X,Y) : -true|\{X = a\}, \{Y <= b\}|true, \\ & q(X,Y) : -\{X <= a\}|\{Y = b\} \rightarrow s(a,b), \\ & q(X,Y) : -true|\{X = a\} \rightarrow r(a,Y)t(a,Y)\} \end{aligned}$$

ここで $s(X,Y)$ の節に注目すると、ボディ部分に出現したサブゴール $t(a,Y)$ が 1 行目の $t(X,Y)$ の節を用いて展開され、P' の 3 行目のようになっている。新たな節はガードが二重に入れ子になっていることに注目されたい。
(例 5 終わり)

プログラム P の節 c に展開規則を適用した結果を $Unf(c,P)$ と表記する。P を NGHC プログラム $\{c_1, \ldots, c_n\}$ とする。P の展開 $Unf(P)$ とは次のように定まる節の集合である。

$$Unf(P) = MIN(\cup_{i=1\ldots n} Unf(c_i, P))$$

P と $Unf(P)$ が成功集合について等価であることは容易に示せる。

定義

P をNGHCのプログラムとする。次のような $P_0, \ldots P_i, \ldots$ を考える。

$P_0 = P$

$P_{i+1} = Unf(P_i)$.

(P_i はすべて等価である。) さらに P_i についてガード付解釈 U_i を次のように定める。

$U_i = \{c \mid c = A : -I_1|O_1, I_2|O_2, \ldots I_n|O_n$

は P_i に含まれる単位節、かつ p は P で定義された述語.$\}$.

このとき、プログラム P について P の展開規則によるセマンティクスとは次のように定まる $U(P)$ である。

$$U(P) = \cup_{i \in \omega} U_i$$

このように定められたセマンティクスが通常の最小不動点によって定まるセマンティクスと等価であることが [22] に示されている。

4.2 Saraswat の CC

先の章までに様々な結果を紹介してきたが、これらの結果が得られる過程で明らかになってきたことのひとつに、単一化に制限を加える方式特に具体化できる変数の制限によって同期 / 制御の機構とすることが、合成的に定義されかつ充分に抽象的で失敗 / デッドロックを定式化できる意味論を与えることを難しくしている原因であるらしい、という観測がある。すなわちある単一化の結果はその両辺に出現する項にあらわれる変数を具体化できるか否かによって決定されるが、並列論理型言語の場合この変数が具体化できるか否かはその単一化と実行環境だけから定めることはできず、その変数が出現した構文的なコンテクストに依存して定まるという問題がある。

Concurrent Prolog を例にとろう。具体化できない変数は**読み出し専用表記** (read only annotation) "?" によって指定される。しかしながら、プログラム本文の出現する "?" のついている変数だけが具体化できない変数というわけではない。実際、実行時に読み出し専用の変数と単一化された変数はそれ自身はプログラムの字面上 "?" がついていなくても、読み出し専用変数と解釈される。すなわち、読み出し専用表記を含む項の単一化の結果は単に現在の束縛の状況だけでなく、計算の動的な履歴に依存して定まる。

GHC の場合は読み出し専用表記の有無の扱いに関する難しさは無いが、変数がゴールに出現したものであるか否かはガード部分に出現する単一化ゴールだけからは判定できず、どちらにしても節の一部の意味を決定する手順として全体を参照する必要性を残しており、合成的に定義可能なセマンティクスを与える上での妨げとなっている。

このことから「受動的に解かれるべき部分」の指定を変数の出現の状況で指定する言語仕様から脱脚し新たな指定の方法を導入することが、より簡潔に合成的に定義されるセマンティクスを与えることのできる言語の導入の鍵といえよう。

例えば [56] では Backus の FP のアイデアを取り入れて論理変数の無い並列言語を提案し、操作的意味論が簡単になるとしている。

ここではこのような事情等を背景として Saraswat によって提案された**並列制約言語系** CC (Concurrent Constraint)[62, 63] について解説する。CC では制約論理型言語の考え方を取り入れているが、このアイデアは [48] で導入されたガード部分に受動的な制約を持つコミット選択型の並列論理型言語 ALPS によって最初に導入された。CC は ALPS やこれまで本稿で扱ってきた言語とは異なり、構文的にはもはや Horn 論理型言語の拡張とは考え難く、実行も項の単一化ではなく制約の解消 (constraint solving) を基本操作としている。しかしながら、そこでのプロセスの軌道と並列実行の形式は、並列論理型プログラムの場合と共通する点も多い。

本節では並列論理型プログラムの一種と考えられる CC のサブセットについて簡単の解説し、その言語の形式的意味論を考える場合の利点について述べる。

CC はその一般的な枠組みでは、具体的な領域の上で定義された言語ではなく、制約系 (constraint system) と呼ばれる抽象的な領域を考え、その各元である制約をプロセスが処理するようなパラダイムとして提案されている。制約系 C は一般には連接 (conjunction) について閉じているものとする。具体的には、例えば項の集合 $Term$ の上での単一化は制約の例である。すなわち代入が個々の制約であり、代入の合成は制約の連接である。

通常の並列論理型言語では変数への束縛情報は、それを具体化したプロセスが他のプロセスに直接転送するものであり、解代入全体を保持 / 管理するような大域的な存在は仮定していない。一方、CC では**貯蔵域** (store) と呼ばれる計算の各時点での中間結果を保持するものを考える。貯蔵域は、計算の結果を制約の連接 (conjunction) の形で保存している。すなわち、貯蔵域の内容は制約の集合によって記述された公理系又は理論である。[48] の ALPS では同様なものを**領域理論** (domain theory) と呼んでいる。直感的にはある時点の貯蔵域の内容が表わす計算結果は、その時の内容に含まれるすべての制約を充たす値の集合、すなわち理論のモデルである。

CC での計算はこの貯蔵域の内容に制約をつけ加えることによって計算を進める。これは、論理型言語において計算途中で得られた解代入にさらに束縛情報をつけ加えることによって計算を進めることに対応する。CC では貯蔵域の内容をアクセスする操作は *ask* と *tell* の二種類が定義されている。いずれの操作も制約を引き数としている。

ask はいわば貯蔵域への読み出しアクセスあるいは条件判定である。その役割は条件文

またはGHCのガードのうちバインディングをつくらないものに近い。*ask(c)*の実行は次のようなものである。引き数となっている制約cが貯蔵域の内容から帰結されれば、すなわち貯蔵域の内容σが制約cを既に充たしているなら実行は成功し、その結果貯蔵域の内容はσのまま変化しない。もし、σがcと矛盾するときは*ask*の実行は失敗する。cがσから帰結もされず、矛盾もしないときには*ask*の実行はブロックされる。すなわち*ask*が成功するための情報がσに不足しているわけである。これらは並列論理型プログラムにおけるガードの成功、失敗、サスペンドに性格対応するものと考えられる。

一方*tell(c)*の実行は次の様なものである。*tell*の引き数cがσと矛盾しない場合は、実行の結果新たな貯蔵域の内容は$\sigma \wedge c$となる。CCの基本となる版ではこの動作はアトミックに実行されるものとしている。σとcが矛盾する場合は、実行は失敗する。この点ではGHCのボディ部分に出現する単一化に対応するものと考えられる。しかしながら、*tell*の実行がアトミックであるという点ではConcurrent Prologに近いといえる。Saraswatはこの*tell*のセマンティクスをeventualに定義しなおせば、GHCがCCの特殊な場合として考えられるとしている。

cはask操作又はtell操作、Aをステートメントとする。CCの基本的な構文では$c \rightarrow A$によって逐次実行、すなわち「cを実行した後にAを実行する」を表わす。計算の各時点の表記は$< A, \sigma >$によって表記する。この場合は現在の貯蔵域の内容がσでありこれから実行されるステートメントがAであることを表わす。$< A, \sigma >$という状態から変数の集合Vを通じて観測可能な動作aを実行した結果状態が$< A', \sigma' >$となることを、$V \vdash < A, \sigma > \stackrel{a}{\longmapsto} < A', \sigma' >$と記述する。

このような記法を用いると、ask/tell演算の実行は次のようなCCS風の推論規則によって記述される。

$$\frac{C \models (\exists)(\sigma \wedge c)}{V \vdash < tell(c) \rightarrow A, \sigma > \stackrel{tell(c)}{\longmapsto} < A, \sigma \wedge c >}$$

$$\frac{C \models \sigma \Rightarrow c}{V \vdash < ask(c) \rightarrow A, \sigma > \stackrel{ask(c)}{\longmapsto} < A, \sigma >}$$

Cはプログラムが定義されている制約系である。CCは基本的にこのような制約を扱う操作を基礎にCCS的な構文要素、すなわちor分岐$+$、並列実行$\|$等を用いてプログラムを記述する言語である。このようなプログラムのセマンティクスについて、[62]で

は、各構文要素のセマンティクスを上のようなCCS風の推論規則で定義した、操作的意味論を与えている。

またここでは詳しくは述べないが、最近では貯蔵域の値に対して容易に定義できる半順序を用いて表示的な意味論を与える試みも行なっている[63]。 従来の並列論理型言語が、効率的な実現の可能性や記述力等に着目して同期/通信等の機構を設計してPrologにつけ加えるような形で提案されているのに対し、 このように新たな言語においては従来の言語に再検討を加え、特にその理論的な明瞭さを当初から尊重して提案されている。

特に注目に値するのは、このアイデアをもとに並列論理型言語について合成的で充分に抽象的なセマンティクスを従来より容易に与えることができることが期待されることである。 実際従来の言語を代入を基礎とするものからask/tellの枠組みへと発展させることにより、意味論についてより強い結果が得られてきている。例えば従来ゴールの連接について合成的なセマンティクスのみが話題になっていたが、[28]ではそれに加えて、制限付きながら節の集合の和についても合成的に定義されるセマンティクスが提案されている。

また従来行なわれてきPARLOG, Concurrent Prolog, GHC等の言語の意味論的な側面からの比較についても、これらの言語をCCの枠組みで整理することにより、新たな進展がみられる[11, 13]。

また本節の最初に述べたNGHCに関連する仕事として、NGHCのガード部分において単一化をtell演算に、一方向単一化をaskに置き換えた言語を考え、これについてやはり完全性を備えた展開規則と展開規則意味論が提案されている[26]。 この言語では展開規則はNGHCで与えられたものより大幅に簡単なものとなっている。これは以下のような理由による。NGHCの場合は展開規則を適用する際に節が強標準形になっていることを仮定していた。その結果NGHCにおける展開規則は、実際の展開と強標準形に変換する操作との組み合わせになっていた。実はこの強標準形を仮定し一般の節を対象にしなかった理由は、並列論理型言語で合成的かつ充分に抽象的な意味論を与えるのが一般に困難であるのと同じ理由、すなわちNGHCがそれまでの並列論理型言語同様に変数の出現によって具体化の可不可を制限する方式を採用していたことによる。 したがって、ask/tell演算を基礎に言語を再構築することによって展開規則はその標準形を得る操作の部分で、より簡単でエレガントなものにすることができたのである。

このように従来の言語の形式的意味論の研究結果を新たな言語の提案に反映させることにより、形式的意味論をはじめとするプログラム言語の理論的研究においてより進んだ結果が得られてきているという事実は、これらの言語の研究が健全な形で進んでいることをうかがわせるものと言えよう。

5 まとめ

並列論理型言語は未だ十年程度の歴史しか持たない若い言語であり、その形式的意味論の研究については、殆ど最近4年以内に注目すべき結果は集中している。[68]では「並列論理型言語のセマンティクスについての研究は(89年初頭においては)いまだ流動的であり、ここではsurveyしない」とある。本稿での解説は現在流動的である研究分野が、どのような歴史的流れに乗って動いているのかを、特にそれぞれの研究結果の位置付けに注意を払いながら述べたつもりである。

謝辞: 本解説について原稿段階より有益なコメントをして下さった國藤室長、田中研究員はじめ富士通国際情報社会科学研究所2)-2)研究室の皆様に感謝します。また本稿は第五世代コンピュータ・プロジェクトの一環として行なわれた調査研究である。

参考文献

[1] K. Apt and M. van Emden, Contributions to The Theory of Logic Programming, J. ACM 29, pp 841-862, 1982

[2] L. Beckman, Towerds a Formal Semantics for Concurrent Logic Programming Language, Porc. of Third Int. Conf. of Logic Programming, pp 335-349 (1986)

[3] J. Bergstra and J. Klop, Algebra of Communicating Porcesses with Abstraction, Theoret. Comp Sci. 37, pp 77-121 (1985)

[4] J. Brock and W. Ackermann, Senario: A Model of Nondeterminate Computation, In Formalization of Programming Concepts, J.Daiz and I. Ramos (ed.), LNCS, Vol. 107, Springer-Verlag, pp. 252-259, (1981)

[5] S. Brooks, C. Hoare, A. Roscoe, A Theory of Communicating Sequential Porcesses, J. ACM, 31, pp 560-599, (1984)

[6] A. Corradini and U. Montanari, Towards A Process Semantics in The Logic Programming Style, Italian- Japanese-Swedish Workshop on Concurrent Logic Programming and Constraint Logic Programming, Pisa, Italy, (1989)

[7] J.W. de Bakker and J.N. Kok, Uniform Abstraction, Atomicity and Contractions in The Comparative Semantics of Concurrent Prolog, Proc. of Int. Conf.on Fifth Generation Computer System 1988, pp 347-355 (1988)

[8] J.W. de Bakker and J.N. Kok, Comparative Metric Semantics for Concurrent Prolog, Theoret. Comp. Sci, vol. 75, No. 1/2, pp 15-43 (1990)

[9] F. de Boer, J.Kok, C. Palamidessi and
J.J.M.M. Rutten, Semantic Models for A Version of Parlog, Proc Of Sixth Int. Conf. of Logic Programming, pp 621-636 (1989)

[10] F. de Boer, J.Kok, C. Palamidessi and
J.J.M.M.. Rutten, Control Flow Versus Logic: A Denotational and Declarative Model for Guarded Horn Clauses, Mathematical Foundation of Computer Sci. Springer LNCS No. 379, pp 165-176 (1989)

[11] F.S. de Boer and C. Palamidessi, Concurrent Logic Programming: Asynchronism and Language Comparison , North American Conf. on Logic Programming '90, pp 175-194 (1990)

[12] F.S. de Boer and C. Palamidessi, A Fully Abstract Model for Concurrent Constraint Logic Languages, to appear in CONCUR '90, Springer LNCS

[13] F.S. de Boer and C. Palamidessi, Modular Embedding, ICLP-90 Pre-Conference Workshop on Semantics of Concurrent Logic Languages, Eilat, Israel (1990)

[14] K. L. Clark and S. Gregory, PARLOG: Parallel Programming in Logic, ACM Trans. on Prog. Lang. Syst., Vol.8, No.1, pp.1-49, (1986)

[15] M. Codish, J. Gallagher, E. Shapiro, Using Safe Approximations of Fixed Points for Analysis of Logic Programs, Proc. of META 88, Workshop on Meta-Programming in Logic Programming, pp 185-197 (1988)

[16] C. Codognet, P. Codognet and M.M. Corsini, Abstract Interpretation for Concurrent Logic Languages, North American Conf. on Logic Programming '90, pp 215-232 (1990)

[17] A. Corradini and U. Montanari, Towards A Process Semantics in The Logic Programming Style, Italian-Japanese-Swedish Workshop on Concurrent Logic Programming and Constraint Logic Programming, Pisa, Italy, (1989)

[18] M. Falaschi, G. Levi, M. Martelli and C. Palamodessi, A New Declarative Semantics for Logic Languages, Porc. of 5th Conf. and Symp. on Logic Programming '88, 993-1005, (1988)

[19] M. Falaschi and G. Levi, Operational and Fixpoint Semantics of A Class of Committed-Choice Logic Languages, Dipartimento Di Informatica, Università Di Pisa, Italy, Techn. Report, January 1988

[20] M. Falaschi and G. Levi, Finite Failures and Partial Computations in Commited-Choice Languages, Proc. of Int. Conf.on Fifth Generation Computer System 1988, pp 364-373 (1988)

[21] M. Falaschi and G. Levi, Finite Failures and Partial Computations in Concurrent Logic Languages, Theoret. Comp. Sci, vol. 75, No. 1/2, pp 45-66 (1990)

[22] M. Falaschi, M. Gabbrielli, G. Levi, M. Murakami, Nested Guarded Horn Clauses: a language provided with a complete set of Unfolding Rules, Proc. of The Logic Programming Conference '89, pp 143-154 (1989)

[23] K.Furukawa, A. Okumura and M. Murakami, Unfolding Rules for GHC Programs, Proc. of the Workshop on Partial and Mixed Computation, New Generation Computing, vol 6, No. 2,3, pp 143-157 (1988)

[24] 淵 一博監修, 古川 康一, 溝口 文雄共編, 並列論理型言語 ghc とその応用, 知識情報処理シリーズ第 6 巻, 共立出版, (1987)

[25] M. Gabbrielli and G. Levi, An Unfolding Reactive Semantics for Concurrent Constraint Programs, to appear as Tech Rep of Pisa Univ.

[26] M. Gabbrielli and G. Levi, Unfolding and Fixpoint Semantics of Concurrent Constraint Logic Programs, Algebraic and Logic Programming, Lecture Notes in Computer Science 463, pp 204-216, (1990)

[27] M. Gabbrielli and G. Levi, A Semantic Reconstruction of Concurrent Constraint Logic Programming, Italian-Japanese-Swedish Workshop on Concurrent Logic Programming and Constraint Logic Programming, Pisa, Italy, (1989)

[28] H. Gaifman and E. Shapiro, Fully Abstract Compositional Semantics for Logic Programs, CS-88-15, Proc of 16the Annual ACM Symp. on Pronciples of Programming Languages,pp 134-142 (1989)

[29] H. Gaifman and E. Shapiro, Proof Theory and Semantics of Logic Programs, Proc. of the IEEE Symp. on Logic in Computer Science, 50-62 (1989)

[30] H. Gaifman, M. Maher and E. Shapiro, Reactive Behavior Semantics for Concurrent Constraint Logic Programs, Proc. of North American Conf. on Logic Programming, pp 553-569 (1989)

[31] H. Gaifman, Semantics of Logic Programs (Advanced Tutorial), Proc. of 7th Int. Conf. of Logic Programming, pp 779-782 (1990)

[32] H. Gaifman, M. Maher and E. Shapiro, A General Setup for Asynchronous Concurrency with Prallel Compsitions, ICLP-90 Pre-Conference Workshop on Semantics of Concurrent Logic Languages, Eilat, Israel (1990)

[33] R. Gerth, M. Codish, Y. Lichtenstein, E. Shapiro, Fully Abstract Denotational Semantics for Flat Concurrent Prolog, Proc. of the IEEE Symp. on Logic in Computer Science, pp 320-333 (1988)

[34] S. Gregory, Parallel Logic Programming in PARLOG, Addison-Wesley, (1987)

[35] C.A. Hoare, Communicating Sequential Processes, Prentice Hall (1985)

[36] J.M. Jacquet and L. Monterio, Comparative Semantics for a Parallel Contexual Logic Language, North American Conf. on Logic Programming '90, pp 195 214 (1990)

[37] G. Kahn, The Semantics of a Simple Language for Parallel Programming, Proc. of IFIP CONGRESS 74, North Holland, pp 471-475 (1974)

[38] G. Kahn and D.B. Macqueen, Coroutines and Network Of Parallel Processes, IFIP CONGRESS 77, North-Holland, pp 993-998 (1977)

[39] J. Kok, A Compositional Semantics for Concurrent Prolog, Proc. of Symp. on Theoretical Aspects Computer Science. Springer LNCS 294, pp 373-388 (1988)

[40] J. Kok, Semantic Models for Parallel Computation in Data Flow, Logic- and Object-Oriented Programming, Ph. D. Thicses of Free University of Amsterdam (1989)

[41] J. Kok, Semantic Models for Parallel Choice in Combnation with Atomicity, ICLP'90 Pre-Conference Workshop on Semantics of Concurrent Logic Languages, Eilat, Israel (1990)

[42] G. Levi and C. Palamidessi, The Declarative Semantics of Logical Read-Only Variables. Proc. of IEEE Symp. on Logic Programming '85, pp 128-137 (1985)

[43] G. Levi, C. Palamidessi, An Approach to The Declarative Semantics of Synchronization in Logic Languages, Proc. of Forth Int. Conf. on Logic Programming, The MIT Press, pp 877-893 (1987)

[44] G. Levi, A New Declarative Semantics of Flat Guarded Horn Clauses, TR-345, ICOT (1988)

[45] G. Levi and C. Palamidessi, Contributions to The Semantics of Logic Perpetual Processes, Acta Informatica 25, pp 691-711 (1988)

[46] G. Levi, Models, Unfolding Rules and Fixpoint Semantics, Proc. of the Fifth Int. Conf. and Symp. on Logic Programming, MIT Press, pp 1649-1665 (1988)

[47] J. W. Lloyd, Foundations of Logic Programming, Springer-Verlag, 1984, 佐藤 雅彦, 森下 真一 訳, 論理プログラミングの基礎, ソフトウェア サイエンス シリーズ, 産業図書 (1987)

[48] M. Maher, Logic Semantics for A Class of Commited Choice Programs, Proc. of Forth Int. Conf. on Logic Programming, The MIT Press, pp 858-876 (1987)

[49] R. Milner, Communication and Concurrency, Prentice Hall (1989)

[50] M. Murakami, Proving Partial Correctness of Guarded Horn Clauses Programs, Logic Programming '87, Lecture Notes in Computer Science no. 315, Springer-Verlag (1988)

[51] M. Murakami, A New Declarative Semantics of Parallel Logic Programs with Perpetual Processes, Proc. of Int. Conf. on Fifth Generation Computer System '88, pp 374-381 (1988)

[52] M. Murakami, Fixpoint Semantics of Guarded Horn Clauses Programs, Italian-Japanese-Swedish Workshop on Concurrent Logic Programming and Constraint Logic Programming, Pisa, Italy, (1989)

[53] M. Murakami, A Declarative Semantics of Flat Guarded Horn Clause for Programs with Perpetual Processes, Theoret. Comp. Sci, vol. 75, No. 1/2, pp 67-83 (1990)

[54] C. Palamidessi, A Theory for Modeling The Synchronization Mechanisms of Concurrent Logic Languages, Italian-Japanese-Swedish Workshop on Concurrent Logic Programming and Constraint Logic Programming, Pisa, Italy, (1989)

[55] R. Ramanujam, Semantics of Distributed Definite Clause Programs, Theoret Comp Sci. 68, pp 203-220 (1989)

[56] Sven-O..lof Nystrom, On The Semantics of Concurent Logic Programming: A Variable-Free Concurrent Language and Its Operational Semantics, Italian-Japanese-Swedish Workshop on Concurrent Logic Programming and Constraint Logic Programming, Pisa, Italy, Uppsala Theses in Computer Science No. 7/89 (1989)

[57] V. A. Saraswat, Partial Correctness Semantics for CP[$\downarrow, |, \&$], Lecture Notes in Comp. Sci., No. 206, pp343-368 (1985)

[58] V.A. Saraswat, Problems with Concurrent Prolog, Tec. Rep. of Carnegie-Mellon University, CMU-CS-86-100 (1986)

[59] V.A. Saraswat, The Language GHC: Operational Semantics, Problems and Relationship with CP($\downarrow, |$), Proc. of Forth IEEE Symp. on Logic Programming, pp 347-358 (1987)

[60] V. A. Saraswat, The Concurrent Logic Programming CP: Definition and Operational Semantics, Proc. of ACM Symp. on Principles of Programming Languages, pp 49-62 (1987)

[61] V. A. Saraswat, A Somewhat Logical Formulation of CLP Synchronization Primitives, Proc. of the Fifth Int. Conf. and Symp. on Logic Programming, MIT Press, pp 1298-1314 (1988)

[62] V. A. Saraswat, Concurrent Constraint Programming Languages, Ph. D. Thises of Carnegie Mellon Univ. (1989)

[63] V. A. Saraswat, Concurrent Constraint Programming Proc. of 17th ACM Symp. on Principles of Programming Languages, pp 232-245 (1990)

[64] V. A. Saraswat, The Paradigm of Concurrent Constraint Programming, Proc. of 7th Int. Conf. on Logic Programming, Tutorial 1, pp 777-778 (1990)

[65] V.A. Saraswat, Fully Abstract Semantics for Concurrent Constraint Programming Languages, ICLP'90 Pre-Conference Workshop on Semantics of Concurrent Logic Languages, Eilat, Israel, (1990)

[66] E. Shapiro, Concurrent Prolog: A Progress Report, Computer, vol. 19, no. 8, pp.44-58, 1986

[67] E. Shapiro (ed), Concurrent Prolog: Collected Papers, vol. 1-2, The MIT Press, 1987

[68] E. Shapiro, The Family of Concurrent Logic Programming Languages, ACM Computing Survays, vol. 21, no. 3, pp 413-510, (1989)

[69] E. Shibayama, A Compositional Semantics of GHC, Proc. of 4th Cof. JSSST, pp 255-258 (1987)

[70] A. Takeuchi, Towards a Semantic Model of GHC, Tech. Rep. of IECE, Comp86-59 (1986)

[71] K. Ueda, Guarded Horn Clauses, ICOT Technical Report, TR-103, ICOT (1985)

[72] 上田, 並列プログラミング言語 GHC の設計思想, 共立出版 ,bit, vol 19, no. 12, pp 4-14 (1987)

[73] 上田, 並列プログラミングと GHC, bit, vol 20, no. 10, pp 83-98 (1988), 井田. 田中共編: 続 新しいプログラミング・パラダイム, 共立出版 (1990)

[74] K. Ueda, Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. Programming of Future Generation Computers, North-Holland, (1988)

[75] K. Ueda and K. Furukawa, Transformation Rules for GHC Programs, Proc. of Int. Conf.on Fifth Generation Computer System '88, pp 582-591 (1988)

[76] K. Ueda and M. Murakami, Formal Semantics of Flat GHC, 平成元年電気・情報関連学会連合大会, 32-3 (1989)

[77] K. Yoshida and T. Chikayama, A'UM - A Stream-Based Concurrent Object-Oriented Language - , Proc. of Int. Conf.on Fifth Generation Computer System '88, pp 638-649 (1988)