(1992. 8. 21)

高並列言語 fleng における型システムの導入

中田 秀基, 田中 英彦
{nakada,tanaka}@mtl.t.u-tokyo.ac.jp
東京大学 工学部

概要

論理型言語を祖先とする Committed-Choice 型言語 (CCL) は、細粒度なデータ駆動プログラミングに適している。しかし、プログラミングのツールとしては、プリミティブな機能しか提供していないため、記述力に問題があることが指摘されている。

従来の CCL の欠点としては、構造データを表現する方法が乏しいこと、クローズ選択時の非決定性が直観に反することなどが挙げられる。

これら欠点を解消するため、完全に単一代入のオブジェクトを導入し、さらにクローズ選択時の不必要な非決定性を排した CCL F を示す。F は、オブジェクトの型の階層を用いてクローズの選択を行なうことで、クローズの継承と多相性を実現する。

Introduction of a Type-system to the Highly-parallel Langauge fleng

Hidemoto Nakada, Hidehiko Tanaka
{nakada,tanaka}@mtl.t.u-tokyo.ac.jp
Faculty of Engineering, The University of Tokyo,
Hongo 7-3-1, Bunkyo, Tokyo, 113 JAPAN

Abstract

Committed-Choice Languages (CCL) are suitable for fine-grained data-driven programming. However, they only provide primitive ways for the programming, so they don't have enough expressive power.

One of the defects of CCL is lacking of structured data. Another is unnecessary non-determinism at choice of a clause. It often againsts programmer's intuition.

In this paper we propose a new CCL F . F adopts an object mechanism for structured data. The objects are single assignment and they have type-hierarchy. F also incorporates novel clause-choice system which takes away unnecessary non-determinism.

We show that introduction of the type-hierarchy and the clause-choice system leads to inheritance of clauses and polymorphism.

1 はじめに

論理型言語を祖先とするCommitted-Choice 型言語(CCL)は、細粒度なデータ駆動型プログラミングに適している。しかし、構造データを表現する方法が乏しいこと、クローズ選択時の非決定性が直観に反することなどの欠点が存在する。

本稿ではこれら欠点を解消するため、単一代入のオブジェクトを導入し、さらにクローズ選択時の不必要な非決定性を排したCCL F を示す。F は、オブジェクトの型の階層を用いてコミットを行なうことで、継承を実現する。

2章で、従来のCommitted-Choice 型言語について概観し問題点を指摘する。3章で、Committed-Choice 型言語に型を導入した F を示す。4章で、型によるクローズの選択に関して詳説する。5章で、本言語で記述した例を示す。

2 Committed-Choice 型言語

2.1 従来のCommitted-Choice 型言語

Committed-Choice 型言語は、並列論理型言語を祖先とする細粒度高並列記述に適した言語であり、Concurrent Prolog [Sha83]、GHC [Ued85]、Fleng [NT88] 等が研究されている。

Committed-Choice 型言語は以下のような特徴を持つ。

- 単一代入の変数を持つ
- プログラムは、プレディケイトの集合で定義される。
- プレディケイトは、同名同アリティのクローズの集合である
- 実行単位はゴールと呼ばれ、ゴールとコミットしたクローズが実行される。
- クローズは、ガードゴールとボディゴールからなり、ガードゴールによってクローズの選択と実行の同期がとられる。あるゴールに対して、ゴール内の変数に値がバインドされていないために、クローズの選択ができない場合にはそのゴールはサスペンドし、その変数がバインドされるのを待つ。
- あるゴールの呼びだしに対して、複数のクローズが実行可能である場合には、クローズのどれが実行されるかは非決定的である。

これらの特徴によりCommitted-Choice 型言語は、変数への値のバインドをガードで待つことによる細かい単位での同期が自然に記述でき、データ駆動実行の記述に適している。

2.2 プロセス指向プログラミング

CCLでは、変数が単一代入であるため、動的に変化する状態を変数に蓄えることができない。これを解決するプログラミングスタイルが、プロセス指向プログラミングである。

プロセス指向プログラミングは、リストをストリームとし、ストリームの要素を一つずつ処理する形でテイルリカーシブなクローズを記述するものである。テイルリカーシブであるため、あるクローズがコミットされると同時に同名同アリティの次の世代のクローズが生成される。この一連のクローズを、一つのアイデンティティをもつプロセスとみなす。このように記述されたプロセスは、ストリーム上のデータに応じて各世代でゴールの引数を変えながら、ストリームを消費する。

下に、ストリームで整数を受けてそれを加算するプロセスをFleng で記述したものを示す。第一引数でストリームをうけ、第二引数の合計値を変化させてテイルリカージョンを行なっている。

```
counter([A|Rest], Sum):-
    Sum2 is Sum + A,
    counter(Rest, Sum2).
```

このプログラミングスタイルの導入により、動的に状態の変化するものを表現することが可能になる。状態を変数に直接蓄えるわけではなく、状態を持つプロセスへのストリームを変数に蓄えれば良いのである。プロセスをオブジェクトと見たて、ストリームにメッセージを載せると考えると、Actor 的なオブジェクト指向プログラミングが可能になる。

2.3 CCL による副作用の記述

単一代入変数をもつ言語では、変数による副作用が記述できない。副作用を持たないことは、並列計算の記述を容易にする反面通常のプログラミングを困難にする。

一般に、並列プログラミングにおいて、副作用が存在しないことは、複数のプロセス間の同期をプログラマが記述しなくて済むことを意味し、高並列言語には有利な点である。反面、副作用によってプロセス間の相互作用を記述することは、プログラマの直観に即しており、副作用を示す記述が可能であることは、プログラマが明示的に慎重にこれを用いる限り、高並列言語においても重要である。

前項の、プロセス指向プログラミングを用いることで、状態をもつものを表現することができるが、これをプログラムの複数の場所から共有することは容易ではない。プロセスへのリファレンスはストリームとなっている変数である。変数は単一代入であるから、プロセスへのリファレンスの共有を単純に変数の共有としてしまうと、そのプロセスに通信できるのは、先にその変数にメッセージをバインドした方だけになってしまう。このため、プロセスを単に変数で共有することでは副作用を記述することはできない。

プロセスへのリファレンスを共有するための方法として、プロセスへのストリームをマージャーと呼ばれるプロセスを用いて分配する方法がとられる。マージャーは二つのストリームを非決定的にマージするプロセスである。これを用いることで、副作用が記述できる。マージャーの非決定性は、ゴールをコミットする際の非決定性に由来する

2.4 従来のCCL の問題点

従来のCCL には、以下の問題点があると思われる。

2.4.1 構造データの取り扱い

Fleng などでの構造データは、2通りの方法で表現される。一つは、アレイやリストに構造を持たせる方法である。以下はFleng で記述されたプログラムの一部である。

```
isMatch(Co, [{ArgNo, CoNo, CoArg}|Next], R):-
        Co = [[{Sym, Args}, CV] | PacketVec],
        ......
```

ヘッド部の [{ArgNo, CoNo, CoArgNo}|NextList] は、リストの中に、3要素のアレイがあることを仮定して、値の取り出しを行なっている。

この方法は、Cで言えば構造体ではなく、すべての構造データを配列で持ち添え字で参照するようなものであり、プログラムは、記述しにくく読みにくいものになる。また、データ構造の一部を変更することで、この構造を用いているクローズすべてを書き直さなければならない。

もう一つは、ゴールの引数に並べる方法である。これは、前述したプロセス指向プログラミングと、深い関係を持つ。プロセス指向プログラミングにおいては、プロセスを形成するクローズのストリームを受けとる引数以外の引数は、いわばプロセスの内部状態を示す変数となる。例えば、下は Fleng のプログラムの一部である。

```
point([xy(Xadd, Yadd)| Rest], X, Y):-
    X1 is X + Xadd,
    Y1 is Y + Yadd,
    point(Rest, X1, Y1).
```

これは、2次元空間上での位置を示すプロセスであり、第一引数のストリームに受けたメッセージに従って、位置を変更する。第2引数、第3引数は、それぞれ、X座標、Y座標を示す内部状態になっている。

このように記述した場合にも、リストなどによる表現と同様に、結局データの意味は記述時の順番によってプログラマが暗黙の内に規定することになる。このため可読性は低く、さらに構造データの仕様が変更された際には、それに関連するゴールをすべて書き直さなければならず、プログラマの大きな負担となる。

2.4.2 余分な非決定性

多くのCCLでは、プレディケイトを構成するすべてのクローズが対等であり、複数のコミット可能なクローズがある場合に、その内のどれがコミットされるかは非決定的である。これは、コミットメントの検査を並列に行なうことを意識した言語の定義であると思われる。しかし、多くの処理系ではコミットメントはシリアルに行なうので、この仕様にはあまり意味がない。

さらに、プログラミングする立場ではこの仕様はしばしば不便である。複数の同名同アリティのクローズを記述する際に、他のどのクローズにもコミットしないときにコミットするデフォルトのクローズの記述が必要になることがある。例えばエラー処理を行なおうとすると、この種の記述は頻繁に必要になる。

以下の例は、メッセージを受けて状態を変えるプロセスである。

```
loop([msg(add(X))|Rest], B, C):-
    .....
    loop(Rest, B1, C1).
loop([msg(A)|Rest], B, C) :- //エラー処理をしたい
    printf("unknown msg: %t\n",[A],_),
    loop(Rest, B, C).
```

2番目のクローズは、プロセスがサポートしていないメッセージがきたらエラー処理を行なおうとしている。しかし、実際にはこのように記述すると、プロセスがサポートしているメッセージ(この場合はadd())がきた時にも、2番目のクローズがコミットしてしまう可能性がある。このように、CCLの非決定性はプログラマの直観に反しプログラマの負担になる。

Concurrent Prolog[Sha83]には、これを解決するために otherwise という機構がある。otherwise 以降に記述されるクローズは、他の同名同アリティのクローズがすべて失敗した場合にコミットする。この記法は1つのプレディケイトにつき1つしか定義できないため充分であるとは言い難い。また、NGHC[FGLM89]ではガード部の条件をネストして記述するため、非決定性をある程度制御することができる。しかし、これを用いるにはプログラマが他の全ての同名同アリティのクローズを意識しなければならず、やはりプログラマの負担が大きい。

3 Committed-Choice 型言語 F

前章に述べたように、CCLは高並列な記述に適した枠組を持っているが、言語としての機能に問題点がある。本稿では、前章で指摘した構造データの表現力とコミット時の不必要な非決定性に着目し、これらの欠点を解消した新たなCCL、Fを示す。

Fでは、スロット名で参照できるCにおけるストラクチャに相当するオブジェクトを導入した。さらに、コミットから不必要な非決定性を取り除き、ゴールに対してもっとも特定的なクローズがコミットされる機構を採用した。タイプ間の階層関係に対して、この機構を適応することにより継承や多相性が実現される。

シンタックスとしてはS式を採用している。

3.1 Committed-Choice 型言語 F の概要

Fは、余分な非決定性を排するため最も特定的なクローズがコミットする機構を採用する。ここで、最も特定的なクローズと言うのは、コミット可能なクローズの内で最もコミット条件が厳しいクローズである。

また、単一代入のオブジェクトを導入する。オブジェクトは、スロット名で参照できるスロットを持つ。オブジェクトは、上位クラスを持つことができる。各クラスは対応する同名の型を持つ。型階層はクラス階層と同型となる。

コミットの条件に型制限を記述することが可能であり、特定的なクローズの選択に型階層を用いることで、クローズの継承を可能にしている。

シンタックスを付録に示す。

3.2 オブジェクト

Fのオブジェクトは、単一代入のスロットを持つ。オブジェクトは単なる構造データであり、多くのオブジェクト指向言語が持つような、オブジェクトに従属するメソッドは持たない。クラス間には階層関係があり、スロット名の継承が行なわれ

る。

3.2.1 オブジェクトの定義

オブジェクトの宣言は以下のように行なう。

(defclass *class-name super-class-list slot-list*)

class-name にオブジェクトのクラス名を記述する。*super-class-list* には、そのオブジェクトのスーパークラス (複数も可能) を記述する。*slot-list* には、そのオブジェクトがもつスロット名を記述する。

オブジェクトを定義することによって、以下の機能を用いることができる。

- ガードにオブジェクトからのデータ抽出および値に関する制約を記述できる
  <*slot-name var-name* >
  <*slot-name value* >
- ガードにオブジェクトの型に関する制約が記述できる。
- オブジェクトのアクセスを行なうクローズが自動的に定義される。
  (*slot-name object var-name*)

例として、2 次元座標系の点を表す point 型のオブジェクトを考えてみよう。

point 型の定義は以下のように行なう。

```
(defclass point () (x y))
```

この定義は、point が上位クラスをもたないこと、x、y という名前のスロットを持つことを示す。

この定義によって、ヘッドに以下の記述が可能になる。

- <x *var-name* >
  : オブジェクトのスロット x の値を変数 *var-name* にとり出す。
- <x *constraint* >
  : オブジェクトのスロット x の値が *constraint* の示す制限にマッチする かどうかをチェックする。
- <y *var-name* >
- <y *constraint* >
- point : オブジェクトの型が point 型 (もしくはそのサブクラス) でなければならないという制約。

また、以下のクローズが自動的に定義される。

- (x *point-object object*):
  point 型オブジェクト *point-object* の スロット x と *object* をバインドする。
- (y *point-object object*)

3.2.2 スロットの継承

F では、複数の直接上位クラスを持つことができる。F オブジェクトが持つのはスロットだけであるので、継承されるものはスロットだけである。あるクラス C に定義されるスロットの名前の集合は、C のすべての上位クラスのスロット名の和集合である。

下のようなクラス構造があったとしよう。このときスロットは図 1 のように継承される。

```
(defclass colored-point (point) (c))
(defclass 3d-point (point) (z))
```

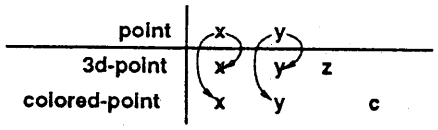

図 1: スロットの継承

3.2.3 オブジェクトの生成

オブジェクトの生成は、以下のシステム述語で行なわれる。

1. new(*type, object*):
   type 型のオブジェクトを生成し、*object* にバインドする
2. new(*type, slot-init-list, object*):
   type 型のオブジェクトを生成し、*slot-init-list* によって イニシャライズを行ない、*object* にバインドする
3. new(*old-object, slot-init-list, object*):
   old-object と同じ型のオブジェクトを生成し、*slot-init-list* によって イニシャライズを行なう。 さらに、*slot-init-list* に指定がないスロットは *old-object* のスロットの値と同じにし、*object* にバインドする

3.3 クローズの選択

F はコミット時に、クローズの引数の型による条件と、引数の値に関する条件を用いて最も特定的なクローズを選択する。

型による条件は型階層においてより近い型を選択するように働く。下のような、クローズがあるとき、p が type1 なら上のクローズが、type2 なら下のクローズがコミットする。

```
(defclause (test (p type1)  ...)
(defclause (test (p type2)  ...)
```

type2 のサブクラスに type3 があったとすると、p が type3 の時には、型の階層においてより近い、type2 を用いた下のクローズがコミットする。

また、F は型の制約以外にその変数の値をコミットの条件にすることができる。これを内部制限と呼ぶ。この制限は、オブジェクトのスロットの値にも、用いることができるので、以下のような記述が可能である。

```
(defclause (test (p point <x 1>)  ...)
```

データの内容に関する条件にも、型に関する制約が記述できる。例えば以下のような記述が許される。

```
(defclause (test (p point <x (n number)>)  ...)
```

3.3.1 多相性

F では、多相性はクローズの選択によって実現される。例として極座標表現の点を考えてみよう。

```
(defclass polar () (r t))
```

r、t は、それぞれ原点からの距離、角度を表す。

この polar と point に対して、以下のように原点からの距離を求めるクローズを定義してみよう。

```
(defclause (distance
          (p point <x x1> <y y1>)  res)
    (mul x1 x1 x2)
    (mul y1 y1 y2)
```

```
    (add x2 y2 sum)
    (sqrt sum res))              // distance 1
(defclause (distance
          (p polar <r r1> <t t1>) res)
    (bind res r1))               // distance 2
```

このように同名同アリティのクローズをオブジェクト複数定義してある場合、(distance p res) というゴールに対して、システムが自動的に適切なクローズを選択しコミットする。

例えば2つの点を与えられて、それぞれの原点への距離の和を返すクローズは以下の様に書ける。このクローズは、point にも polar にも適用できる。

```
(defclause (near-by a b c)
    (distance a da)
    (distance b db)
    (+ da db c))
```

このように、同名同アリティのクローズの中の適切なクローズが選択されることで、多相性が実現される。

3.3.2 クローズの継承

クローズの継承も同様に、クローズ選択によって行なわれる。色つきの2次元の点 colored-point について考えてみよう。このクラスは、point のサブクラスであり、新たに c というスロットを持つ。

```
(defclass colored-point (point) (c))
```

この colored-point に関して全節と同様に原点からの距離を考えてみよう。

colored-point である p を含むゴール (distance p res) があったとしよう。

このゴールに対して最も特定的なクローズは、前節で point に対して記述されたで定義した distance 1 になる。このクローズは、そのまま colored-point にも用いることができるので distance を再定義する必要はない。

この様に、最も特定的なクローズをコミットする機構を用いることで、継承が実現できる。

3.4 システム定義の型

F には、シンボル、数字、リストセル、ベクタがシステムによって用意される。これらのシステム定義のオブジェクトは、それぞれ、symbol、number、cell、vector の型を持つ。これらの型は上位型を持たない。

リストセルは、car、cdr という2つのスロットを持つオブジェクトであると考える。スロットとしては、ベクタに関してはインデックスをスロット名として持つオブジェクトであると考えられる。

```
(defclause (test [1|r]) ...)
(defclause (test {a b c}) ...)
```

は、それぞれ以下のように記述するのと同等である。

```
(defclause (test (x cell <car 1> <cdr r>)) ...)
(defclause (test (x vector <1 a> <2 b> <3 c>)) ...)
```

4 最も特定的なクローズの選択

F はコミット時に、最も特定的なクローズを選択する。クローズに記述できる条件は、引数の型による条件と、引数の値に関する条件である。

まず、型による条件について述べる。型による条件は、前章でも述べたように、指定した型とクローズ選択において、型階層においてより近い型を選択するように働く。類似のシステムに、型階層を用いてメソッドを選択する CLOS[ea87]、Prolog の拡張である PAL[赤間87] がある。

複数の引数に対してメソッドを選択する際には、引数の中の左を優先した順序による選択を行なう。

例えば type1 をベースとする type3 < type2 < type1 の順序の型階層があったとしよう。

ここで、以下のようにクローズの定義があるとする。

```
(defclause (test (x type2) (y type3))
                              ...)  // クローズ 1
(defclause (test (x type1) (y type3))
                              ...)  // クローズ 2
(defclause (test (x type1) (y type2))
                              ...)  // クローズ 3
(defclause (test (x type2) (y type1))
                              ...)  // クローズ 4
```

引数の型がそれぞれ、[type3 type3] である時にはクローズ 1 が、[type2 type2] によである時にはクローズ 4 が選択される。

型による選択機構を導入する際に問題になるのは、CCL ではクローズを決定しようとする時に引数の値が定まっているとは限らないことである。CCL は、クローズの決定にゴール内の変数の値が必要である時には、その変数がバインドされるのをサスペンドして待つが、そうでない場合は変数がバインドされるのを待たずに、コミットしなければならない。

例えば以下のようなクローズがあったとしよう。このクローズのみが存在する場合には、第一引数がバインドされていなくてもサスペンドする必要はない。

```
(defclause (test x (y type1))  ...)
                                    // クローズ 5
```

しかし、このクローズの他に上記のクローズ1～4のいずれかがあれば、クローズを特定するために、サスペンドが必要になる。このような、型指定がされていない変数は最も優先順が低い型指定がされているとみなして実行する。この最も優先順が低い指定子を any とする。クローズ5は、下のクローズ6と等しい。

```
(defclause (test (x any) (y type1))
                              ...) // クローズ 6
```

ある引数に関して型指定があるクローズが存在するときは必ずサスペンドして待てば良いと言うものでもない。下に示すのは、ストリームをマージするマージャーである。

```
(defclause (merge [x | list]  y r)
      (bind r [x | r1])
      (merge list y r1))              // merge 1
(defclause (merge x [y | list] r)
      (bind r [y | r1])
```

```
      (merge x list r1))                    // merge 2
```

このプレディケイトは、一つ目のクローズでは第2引数2つ目のクローズでは第1引数を、それぞれ、バインドされるのを待たずにコミットすることを期待して書かれている。なぜなら、引数がバインドされるのを待つということは、第1引数のストリームに値が到着しており、結果のストリームにこの値を出力することが可能であるにも関わらず、第2引数の到着を待つことを意味するからである。しかし、上記のような方針を用いる限りこのクローズは、常に第1引数を待ってしまい、従って2番目のクローズはコミットされずマージャーとして機能しない。2.3で述べたように、このような非決定性はCCLのプログラミングに副作用を導入するために非常に重要であるので、この非決定性を取り除いてはいけない。

F ではこのような、非決定性を持つクローズを記述するためにvarという疑似型指定子を導入する。この型指定子は、他の全ての型指定に優先する。また、この指定子がある時には、優先順が高いクローズを決定するためにバインドを待つことが必要な場合も、すぐにサスペンドせず他の候補を探す。この指定子を用いるとマージャーは以下のように書ける。

```
(defclause (merge [x | list] (y var) r)
      (bind r [x | r1])
      (merge list y r1))                    // merge 1
(defclause (merge (x var) [y | list] r)
      (bind r [y | r1])
      (merge x list r1))                    // merge 2
```

この機構に関しては4.1で詳しく述べる。

次に、引数の値に関する内部条件について述べる。データの値に関する条件として、オブジェクトが記述できる。オブジェクトは複数のスロットを持つため、いくとおりもの条件の与え方が可能である。これらの間にも優先順位を設ける必要がある。これに関しては4.2で述べる。

以下のクローズのように、型の制約では等価であるが、値に関する制約が異なるクローズがある時にも、最も特定的なクローズがコミットされなければならない。

```
(defclause (test (p point <x 1>)  ...)
(defclause (test (p point <x r>)  ...)
```

また、以下のように型の条件による優先度と、内部条件による優先度が食い違うことがある。この場合は、型による条件を優先する。この例では下のクローズがコミットする。

```
(defclause (test (p point <x 1>)  ...)
(defclause (test (p colored-point)  ...)
```

クローズ選択の方法の詳細な定義は次節以降で行なう。

4.1 コミットするクローズの決定

コミットするクローズの決定に関わる要素は以下の二つである。

- ゴールの引数の型の条件
- ゴールの引数の値の条件

引数の型に関しては、それぞれ型の優先順位リストを求める必要がある。型階層は、型が複数の上位クラスをもつので、木構造ではなく束をなす。この束である型階層から、それぞれの型に関して直列なタイプ優先リストを作成しなければならない。これに関しては4.4で述べる。

引数の値の条件は、条件を付けられたスロットのオブジェクトでの位置によって、重み付けられる。詳細は4.2で述べる。

コミットするクローズの決定は、以下のように行なわれる。同名同アリティのクローズの集合すべてを C とし、注目する変数 v を第一引数とし、また、varがついた変数に関する処理を実行していることを示すフラグ f を偽、サスペンドする対象の変数プール p は空にして以下を呼び出す。

1. C の要素が一つしかない場合、8へ。
2. C を v の型制限で分類する。
3. anyに関する型制限しかない場合は、v を次の着目変数にして1に戻る。次の着目変数がない場合は8へ。
4. varに関する型制限がある場合は、v を次の着目変数にし、C を新たにその型制限をもつ集合とし f を真にし、1を呼び出す。次の着目変数がない場合は8へ。
5. v がバインドされていない時は、v を p に加え、
   (a) f が偽の時は p の要素すべてに対してサスペンドする
   (b) f が真の時には、f を偽にし、f を真にした場所に戻り 実行を続ける。
6. v がバインドされている時は、v の型の優先リストに従って、分類した C をソートする。この結果を $C_1 \sim C_n$ とすると、まず i を1として順に以下を繰り返す。
   (a) C_i の要素を内部制限の優先順でグループ分けしソートする。これを D_j としそれぞれに順に v のスロットを順に新たに v とし、D_j を C として1を呼び出す。
7. ここまで、コミットせずに来た場合は呼び出されたところに戻る。
8. その C のすべてに対してコミットを試みる。コミットできなければ呼び出されたところに戻る。

ここで、次の着目変数とは、その時点で v がゴールの引数であれば、次の引数をしめし、オブジェクトのスロットであれば、次のスロットを示す。

もし、この過程でコミットするクローズが決定しない場合は、コミット可能なクローズが存在しないので、そのゴールは実行されない。この様にコミットの過程を行なうことで、不必要なサスペンドを行なわずにもっとも特定的なクローズを求めることができる。

4.2 内部制限によるクローズの優先順位

例えば、以下のような二つのクローズがあったとしよう。

```
(defclause (test (p point <x 1>)) ..) //A
(defclause (test (p point <y 2>)) ..) //B
(defclause (test (p point <x 1> <y 2>)) ..) //C
```

これに対して、x = 1, y = 2 であるポイントpを含んだゴール(test p)があるとしよう。このゴールに対しては、上記のクローズはすべてコミットすることが可能であるが最も下のクローズが発火することが望ましい。このため、オブジェクトの内部制限に関しても優先順位を設定する必要がある。

優先順位は以下のように定める。

1. オブジェクトのスロットの総数をMとする。
2. 各クローズにおいて、内部制限が存在するn個のスロットS_iのオブジェクト内での順番をN_iとするとき、各クローズの優先度を

$$\sum_{i=0}^{n} 2^{M-N_i}$$

とする。

3. 優先度の大きい順に優先順位をつける。

各オブジェクト内のスロットの順番は4.3で述べる。

上の例の場合、xの順番は1yの順番は2で、総数Mは2である。従って優先度はそれぞれ、2、1、3となり、優先順位は、C、A、Bの順になる。

4.3 各オブジェクトにおけるスロット名の順序

4.2で述べたように、Fでは各オブジェクトにおけるスロット名の順番が重要になる。

スロット名の順番は、以下のように定義する。

- 上位クラスのスロットは、下位クラスのスロットよりも、順番が早い。
- 各クラスのスロット定義でより左のスロットが順番が早い。

これを実現するためには、以下のアルゴリズムを用いる。

1. 直接上位クラスの完全なスロットリストを得る。
2. 得たリストのセットに対して直接上位クラスの優先順に以下を繰り返す。
   (a) 得たリストを順にスロットリストに加える。
   (b) ただし、すでにスロットリストに存在するスロットは加えない
3. 新たにそのクラスで導入されたスロットを最後に加える。

例として3.2.2で定義した、colored-point、3d-pointを用いた3d-colored-pointを見てみよう。

```
(defclass 3d-colored-point
          (colored-point 3d-point) ())
```

3d-colored-pointのスロットは図2のようになる。

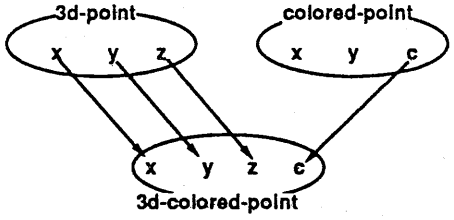

図 2: 3d-colored-point のスロット順

4.4 型優先リストの作成

型は複数の上位型を持つことができる。従って、型の階層は木構造ではなく束構造である。

型優先リストの作成には、CLOSと同様に、super-class-list内の順序を用いたトポロジカルソートを用いる。

あるtypeAに関する優先リストの作成は以下のように行なわれる。

1. typeA及びtypeAのすべての上位型の集合をSとする。
2. Sのすべての要素に関して、そのクラス宣言から順序対の集合を導き、これらの和をRとする。例えば以下のようなクラス宣言があったとしよう。

```
(defclass typeA (typeB typeC) ())
```

このとき、以下のような順序対を得る。

```
(typeA typeB) (typeB typeC)
```

同様にして、typeB、typeCの定義からも順序対集合を導くと、これらの順序対集合の和は(もしクラス定義に誤りがなければ)半順序をなす。

3. 優先順位Aを空にし、以降の処理をSが空になるまで繰り返す
   (a) Sから、Rの要素の順序対で他の型が先行しない型Cをみつけだし、優先順位Aの最後に加える。
   ここで、複数のCの候補が得られた場合には、すでに計算されたAの中で、より後に直接の下位型を持つものを選択する。
   (b) Sから、Cを取り除く。
   (c) Rから、Cを含む順序対をすべて取り除く。

例えば、図3のような型階層があったとしよう。

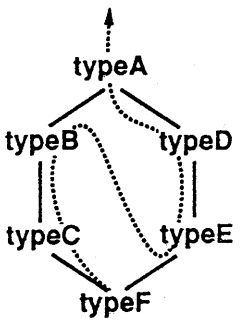

図 3: 型の階層

ここで、typeFに関する型優先リストは、

typeF, typeC, typeB, typeE, typeD, typeA

の順になる。

5 例

CCLではプロセス指向プログラミングが一般的である。ここでは、Fがプロセス指向プログラミングをいかに支援するかを示す。

移動する点の位置を保持し、メッセージに応じて位置を変えるプロセスを考えてみよう。これにはpoint型のオブジェクトを用いる。

```
(defclause (point-loop
        [{add_x x1}| rest] (p point <x x2>))
    (point-loop rest p1)
```

```
    (add x1 x2 x3)
    (new p [['x x3]] p1))
(defclause (point-loop
        [{add_y y1}| rest] (p point <y y2>))
    (point-loop rest p1)
    (add y1 y2 y3)
    (new p [['y y3]] p1))
```

このプロセスは add_x、add_y というメソッドを受けとり、その分だけ保持している点の位置を移動するプロセスである。ここで、プロセスの仕様がかわり、新たに点に色を付加し、その色の変更も行なうことが可能なプロセスにしなければならなくなったとしよう。

従来の CCL では、プロセスを構成する全てのクローズを書き直すことが必要である。しかし、F では point の定義を変え、新たに色を示すスロットを付け加えても、これらのクローズには変更は必要ない。

さらに、point の定義を変更せずにサブクラスを定義することで仕様に対応することもできる。colored-point を 以下のように、point のサブクラスとして定義する。

```
(defclass colored-point (point) (c))
```

すると、この colored-point にも、上記のプロセスを構成するクローズが適用できるので、add_x、add_y に関しては再定義の必要はない。

色の変更に関する下のようなクローズを定義するだけで、新たな仕様を満たすことができる。

```
(defclause (point-loop [{c c1}| rest]
                       (p colored-point))
    (point-loop rest p1)
    (new p [['c c1]] p1))
```

このように、オブジェクトがデータを抽象化しているため、仕様変更に柔軟に対応することができる。また、階層を利用した継承をもちいることで、コードの共有が行なえる。

6 おわりに

従来の CCL の欠点を解消するため、完全に単一代入のオブジェクトを導入し、さらにクローズ選択時の不必要な非決定性を排した CCL F を示した。F は、オブジェクトの型の階層を用いてコミットを行なうことで、多相性、継承を実現する。これらの特徴はプロセス指向プログラミングにおいても有効である。

今後は、プロセス指向プログラミングをサポートするためのマージに代わる機構の導入、シンタックスの改良などを行なう予定である。

参考文献

[ea87] D. G. Bobrow et al.:. Common lisp object system specification. Technical report, ANSI Draft X3 Document 87-003, 1987.

[FGLM89] Moreno Flaschi, Maurizio Gabbrielli, Fiorgio Levi, and Maski Murakami. Nested guarded horn clauses: a language provided with a complete set of unfolding rules. In *Logic Programming Conference '89*, pp. 143–154, 1989.

[NT88] M. Nilsson and H. Tanaka. Massively parallel implementation of flat ghc on the connection machine. In *Int. Conf. on Fifth Generation Computer Systems*, pp. 1031–1040, 1988.

[Sha83] E. Shapiro. A subset of concurrent prolog and its interpreter. TR 003, ICOT, 1983.

[Ued85] K. Ueda. Guarded horn clauses. TR 103, ICOT, 1985.

[赤間 87] 赤間清. Pal: 継承階層を扱う拡張 prolog. 情報処理学会論文誌, Vol. 28, No. 4, pp. 322–329, Apr. 1987.

A F のシンタックス

class-definition:
　　(**defclass** *class-name*
　　　　(*super-classes*)(*slot-list*))
super-classes:
　　null
　　class-name super-classes
slot-list:
　　null
　　slot-name slot-list
clause-definition:
　　(**defclause** *head bodys*)
head:
　　(*clause-name h-exps*)
bodys:
　　null
　　body bodys
body:
　　(*clause-name exps*)
exps:
　　null
　　exp exps
exp:
　　var
　　symbol
　　number
　　list-exp
　　vector-exp
list-exp:
　　[*exps*]
　　[*exps* | *exp*]
vector-exp:
　　{ *exps* }
symbol:
　　' *string*
h-exps:
　　null
　　h-exp h-exps
h-exp:
　　exp
　　(*varname typename specifiers*)
specifiers:
　　null
　　specifier specifiers
specifier:
　　< *slot-name h-exp* >
class-name:
　　string
clause-name:
　　string
slot-name:
　　string