プログラミング－言語・基礎・実践 12－10
（１９９３．５．２８）

宣言型計算モデル

赤間 清

北海道大学 工学部 情報工学科
札幌市北区北 13 条西 8 丁目

関数モデルでは式の評価が計算であり、書換えモデルでは項の書き換えが、また論理モデルでは論理式の証明が計算であるとみなされている。
本論文では、これに対して、宣言型計算モデルという新しい計算モデルを提案する。そこでは、「計算とは宣言的プログラムのプログラム変換である」と考える。宣言的プログラムとは、あるオブジェクト空間の元を atom として作られる節の集合である。
適切なオブジェクト空間を選択することによって、項書換えシステム、制約論理型言語、タイプつきユニフィケーション文法を含む、数多くのプログラミング言語や知識表現系における計算の基礎が宣言型計算モデルとして統一的に議論できる。宣言型計算モデルは、プログラム変換、抽象解釈、タイプ推論などを議論するためにも適している。

Declarative Computation Model

Kiyoshi Akama

Department of Information Engineering, Hokkaido University
kita13, nishi8, kita-ku, Sapporo, 060 Japan

In the functional computation model, the computation is regarded as the evaluation of expressions. In the rewriting computation model, the computation is considered as the rewriting of terms. In the logical computation model, the computation is the proof of formulae.
In this paper we propose a new computation model, called "declarative computation model", where the computation is regarded as the program transformation of declarative programs. A declarative program is defined as a set of "clauses" consisting of atoms in an object space.
By constructing appropriate object spaces, many programming languages and knowledge representation systems, including term rewriting systems, constraint logic programs and typed unification grammars, are formalized as instances in this model. The declarative computation model is useful not only for discussing usual ground computations, but also for constructing a theory of higher level computations, such as nonground computation, program transformation, abstract interpretation and type inference.

1 はじめに

計算は、関数モデル、書換えモデル、論理モデル[1]などの計算モデルに分類されて説明されている。関数モデルでは関数の評価が計算であり、書換えモデルでは項の書き換えが、また論理モデルでは論理式の証明が計算であるとみなされている。このように、計算とは千差万別であり、それらの計算をひとまとめに特徴づける概念は提示されていない。

本論文では、宣言型計算モデルという新しい計算モデルを提案する。そこでは、「計算とは宣言的プログラムのプログラム変換である」と考える。宣言的プログラムとは、オブジェクト空間の元を用いて作られる節の集合である。適切なオブジェクト空間を構成することによって、項書換えシステム、制約論理型言語、純 Prolog 、純 Lisp 、タイプつきユニフィケーション文法を含む、数多くのプログラミング言語や知識表現系の基礎が宣言型計算モデルの枠内で統一的に議論できる。

宣言型計算モデルは、普通の ground な計算（具体的な値の計算）だけでなく、より高次の計算、たとえば nonground な計算（データ構造に変数を含む計算）、プログラム変換、抽象解釈、タイプ推論などの理論の構築にも適した新しい計算モデルになっている。

2 計算概念の拡大と計算モデル

2.1 計算とは何か

計算とは、いったい何であろうか. この問いに対する最初の画期的な回答は、「Turing 機械で計算できるものが計算である」という Turing の提唱であろう。そこでの計算は、部分関数を「状態の遷移」に基づいた方法で実現するものである。この考えは、von Neumann 型計算機や手続き型言語に受け継がれた。

2.2 計算モデル

しかし、部分関数を「状態の遷移」で実現するという考えは、複雑で高度なプログラムを構築する立場から見ると限界があることがわかってきた。そしてそれに束縛されず、人間の思考への適合しやすさ、または理論対系の構築しやすさなどの観点から、新しい計算の概念が提案された。たとえば、関数型言語の世界では、「計算＝式の評価」とみなされた。また、書換えシステムでは、「計算＝項の書換え」であり、論理型言語の世界では、「計算＝論理式の証明」と把えられた。計算を理論化するこれらの体系を計算モデルと呼び、上記の計算モデルを、それぞれ、関数モデル、書換えモデル、論理モデルと呼ぶことにしよう [10]。

2.3 計算概念の拡大

これらの計算モデルに基づいた研究は、計算の概念をそれぞれ独自の方向（評価、書換え、証明など）に拡大すると同時に、共通の方向にも拡大した。

最初に考えられた計算は、具体的な入力から具体的な出力を得るもので、ここでは、それを「ground な計算」と呼ぼう。それは Turing が、計算を部分関数の実現と考えていたのと同様の意味での計算である。これに対して、Prolog の差分リストなどに見られる、変数を含んだデータ構造の操作は、「nonground な計算」と呼ぶことができる。

同じ計算結果を出すプログラムにも効率の良いものと悪いものがある。与えられたプログラムを、より効率の良いものに変更するというアイディアは、プログラム自体を計算の対象にするもので、**プログラム変換**という新しい種類の計算を生み出した。そのために役立つ技術の 1 つに**抽象解釈**がある。これは、もとのプログラムを抽象化して、それが引き起こす抽象レベルの計算の結果からもとのプログラムの情報を得るものであり、抽象的なプログラムと抽象的な計算の必要性が認識された。また、プログラムにタイプをつけて、その情報をもとにして推論し、プログラムの誤りを発見したりする**タイプ推論**もまた、新しい種類の計算ということができる。

2.4 計算モデルの統合

計算モデルは関数モデルや論理モデルなどのように、いろいろな方向に発展しているにもかかわらず、それらの共通点は少なくないことが認識されている。たとえば、プログラム変換の理論は、関数モデルで最初に研究され、それが論理モデルの世界にも導入された。しかしその導入は、直観を基盤として両者の類似点を手掛かりになされたもので、決して両者の間に数学的に厳密な関係を確立してなされたわけではない。関数モデルと論理モデルは共通点はあるが異なるものであると認識されている。しかしもし

[1]本論文では、制約などを扱わない基礎的な論理プログラミングの範囲を指すものとする。

それらの計算モデルを基礎づける共通の数学的枠組があれば、このような研究はずっと円滑になるだろう。同様のことが、タイプ推論や抽象解釈についてもいえる。

計算とは何かをよりよく解明するためにも、各種の計算モデルを統一的な視点から論じることのできる理論的な枠組が必要であろう。それは、各計算モデルの最大限の共通性と差異を数学的にはっきり示すものであることが望まれる。またそれは、いくつかの計算モデルによって重要性を見い出された高次の計算（プログラム変換、抽象解釈、タイプ推論など）を有効に議論できるための構造を十分に備えていることが望ましい。

関数モデルや論理モデルなどの計算モデルの共通の基盤といえば、現在知られているのは、計算を部分関数の実現の問題に符合化して計算可能性を論じる Turing の枠組（と同等のレベルのもの）しかない。しかし、Turing の枠組で測れば、すべての意味あるプログラミング言語は等価となる。Turing の枠組は、各計算モデルの共通性を 1 つ提示しているが、それらの差異や高次の計算を適切に議論できる枠組にはなっていない。

3 宣言型計算モデル

3.1 計算＝宣言的プログラムの等価変換

我々は、多くの重要な計算を統一的にとらえ、プログラム変換、抽象解釈、タイプ推論などを議論するためにも適した新しい計算モデルを提案したい。それは、計算を「宣言的プログラムの等価変換」とみなすモデルで、宣言型モデルと呼ぶことにする。

宣言的プログラム[2] は、あるオブジェクト空間の元を用いて作られる節の集合であり、そのすべてのプログラムに宣言的意味が与えられ、それを保存するプログラム変換が議論できる。

これによって、項書換えシステム、制約論理型言語、タイプつきユニフィケーション文法をふくむ、数多くのプログラミング言語や知識表現系の基礎が宣言的計算モデルの枠組で統一的に議論できる。

宣言型計算モデルは、高次の計算を議論するためにとくに有効と考えられる。宣言型計算モデルは、計算をプログラム変換と見なしており、プログラム変換の議論が中心の理論構成となっている。また、宣言的プログラムの基礎となるオブジェクト空間は、表現力が高く、抽象的なオブジェクトの空間やタイプがついたオブジェクトの空間を明快にかつ容易に構築できる。宣言型計算モデルでは、既存の計算モデルとちがって、はじめから抽象プログラムやタイプ付きプログラムをその第一級市民として認めている。これらは、これまでの計算モデルが生み出した多くの研究成果を自然に継承し、さらに発展させるために有効であると考えられる。

3.2 Specialization Structure

宣言的プログラムの節を構成するオブジェクトの空間は specialization structure によって数学的に特徴付けることができる。specialization structure は宣言的プログラムの理論[3] の中で最も基本的な構造の 1 つである。

定義 1 *specialization structure* とは次の条件を満たす 3 項組$< \mathcal{A}, \mathcal{S}, \mu >$ である。

(1) $\mu : \mathcal{S} \rightarrow partial\_map(\mathcal{A})$

(2) $\forall s_1, s_2 \in \mathcal{S}, \exists s \in \mathcal{S} : \mu(s) = \mu(s_1) \circ \mu(s_2)$

(3) $\exists s \in \mathcal{S}, \forall a \in \mathcal{A} : \mu(s)(a) = a$

$\mathcal{S}$の元を *specialization* と呼ぶ。$\mathcal{A}$ の元を 論理オブジェクト[4] (節のトップレベルでは *atom*[5]) と呼ぶ。

$\mathcal{S}$ の元 s が $\mathcal{A}$ の元 a に適用可能[6] であるとは、$(a, b) \in \mu(s)$ なる $b \in \mathcal{A}$ が存在することである。

3.3 宣言的プログラム

specialization structure $\Gamma =< \mathcal{A}, \mathcal{S}, \mu >$ 上の宣言的プログラムを定義する。

定義 2 $\Gamma =< \mathcal{A}, \mathcal{S}, \mu >$ 上のプログラム節とは, $\mathcal{A}$ の元 $H, B_1, \cdots, B_n$から作られた $H \leftarrow B_1, \cdots, B_n$ $(n \geq 0)$ の形の式である。Γ 上の宣言的プログラムとは、Γ 上のプログラム節の（任意の）集合である。

[2]以前の論文で一般化論理プログラム（Generalized Logic Program）と読んでいたものをここではこう呼ぶことにする。

[3]以前の論文で一般化論理プログラムの理論（GLP の理論）と読んでいたものをここではこう呼ぶことにする。

[4]オブジェクト指向のオブジェクトと区別する意味で、論理オブジェクトと呼ぶが、簡単にオブジェクトと呼ぶこともある。

[5]通常の論理プログラムの理論と違って、atom (atomic formula) と 項を数学的には区別しない。論理オブジェクトは、引数にも atom にもなれる。

[6]specialization は伝統的な論理プログラムの理論での代入にあたる。代入は常に適用可能である。しかし、適用可能でない specialization の存在を認めないと、CLP などを包含できない。

プログラムというのは、普通は「実際に動く」ことを前提とする。しかし宣言的プログラムは、純粋に数学的な概念であり、「実際に動く」必要はない。プログラム変換によって数学的な意味で変化が議論できればよい。宣言的プログラムは、完全に宣言的な記述であり、無限の節からなるものも許され、計算可能性にはこだわらない。これに対して、論理プログラムは制御（cut など）を含み、実際に動作することを前提としており、宣言的プログラムの位置付けとは異なることに注意する必要がある。

3.4 宣言的意味論

P を $\Gamma =< \mathcal{A}, \mathcal{S}, \mu >$ 上の宣言的プログラム とする。$\mathcal{A}$ の任意の部分集合 $\mathcal{G}$ を選び、これを Γ の解釈領域と呼ぶ。すると、宣言的プログラムの場合にも通常の論理プログラムの場合と同様の意味論が構築できて、プログラム P に対して、

$$M_P = lfp(K_P) = K_P \uparrow \omega$$

によって $\mathcal{G}$ の部分集合が定められる。これを P の宣言的意味と呼び $\mathcal{M}(P)$ と書く。この理論は論理モデルの場合と外形的には類似しているが、

- モデルの概念を新しくしている。意味論に、述語や関数という概念は出現しない。
- immediate consequence transformation T_Pではなく、$K_P = T_P + I$ を用いている。
- より広い範囲のプログラムに適用可能である。

などの違いがある。

3.5 プログラムの等価変換と unfold 変換

$\mathcal{M}(P)$ を基礎とすれば、それを保存する変換を考えることができる。それが 宣言的プログラムの等価変換である。等価変換のなかで、unfold 変換がとくに重要である。

宣言的プログラムの unfold 変換の定義は、それを構成する unifier や resolvent の定義とともに、論理モデルにおける定義とは少し異なる。一般的な理論にするために、基本的な概念の定義についても変更が必要である。unfold 変換や unifier 、resolvent の定義については紙面の都合上割愛する。

3.6 unfold 変換の健全性と完全性

プログラム P_1 が unfold 変換によって プログラム P_2 に変換されるとき、その unfold 変換が健全かつ完全であるのは、P_1 と P_2 の宣言的意味が等しいとき、すなわち、$\mathcal{M}(P_1) = \mathcal{M}(P_2)$ が成り立つときである。宣言的プログラムの理論では、unifier 集合の健全性と完全性が定義され、健全かつ完全な unifier 集合を用いて unfold 変換すれば、その unfold 変換は健全かつ完全になることが証明されている。

3.7 スキーマ理論

宣言的プログラムの理論は、オブジェクト空間と呼ばれるプログラムの基礎空間 X をパラメータとして持つ**スキーマ理論** $\mathcal{T}(X)$ であり、X を変えることによって、種々のプログラミング言語や宣言的知識表現系の基礎を統一的に議論することができる。CLP の理論もまた、領域 X をパラメータとするスキーマ理論 CLP(X) であるが、宣言的プログラムの理論は、CLP の理論よりも広範な対象を明快に扱うことができ、CLP もまた、宣言的プログラムの一種としてすでに定式化されている [5]。宣言的プログラムの理論のパラメータであるオブジェクト空間は、unification や 推論（resolution）などを決定するという意味でも CLP のパラメータ X よりも根源的なパラメータであるといえる。

いろいろなプログラミング言語に宣言的プログラムの理論を適用するためには、適切なオブジェクト空間を構成し、各プログラムの動作が、対応する宣言的プログラムの等価変換と一致することをいえばよい。こうして宣言的プログラムとして認定できれば、宣言的プログラムの理論が提供する意味論や unfold 変換の理論などを使って明快に議論することができる。これは新しい知識表現などを構想し、その理論や処理系を作る時にとくに有用である。

4 Prolog の場合

ここでは pure Prolog について考える。

4.1 Prolog のオブジェクト空間

Prolog を S 式シンタックスで考える。Prolog では変数の入った S 式を使う。基本要素として、<アトム>と<変数> が区別されているものとする。

<アトム> = x | y | ab | F2 | 2 | 3 | 54 | 102 | ⋯

<変数> = * | ** | *a | *F2 | *Y | *3 | *xyz | …

そのとき、<Prolog S式> を

<Prolog S式>
= <アトム>
| (<Prolog S式> . <Prolog S式>)
| <変数>

と定義する。Prolog を 宣言的プログラムとみなすためのオブジェクト空間のオブジェクト集合 $\mathcal{A}_P$ は <Prolog S式>そのものである。すなわち、

$\mathcal{A}_P$ = <Prolog S式>

Prolog のオブジェクトに作用している specialization は変数を任意の<Prolog S式>で置き換える代入である。

$\mathcal{S}_P$ = { <変数>/<Prolog S式>, … }

代入がオブジェクトにどう作用するかは明らかである。その写像を $\mu_P : \mathcal{S}_P \rightarrow map(\mathcal{A}_P)$ とするとき、$\Gamma_P =< \mathcal{A}_P, \mathcal{S}_P, \mu_P >$ が Prolog を宣言的プログラムとみなす場合のオブジェクト空間となる。

4.2 SLD 導出 と unfold 変換

論理モデルにおける計算は SLD 反駁である。一方、宣言型モデルでは、計算は unfold 変換などからなる等価変換であると考える。

論理モデルでは、たとえばappend プログラム：

```
(append () *y *y) ← .
(append (*car . *cdr) *y (*car . *app))
    ← (append *cdr *y *app).
```

に対して query (append (1 2) (3) *X) が与えられたとき、SLD 反駁によって、goal：

```
← (append (1 2) (3) *X).
```

に対して変形を加え、空 goal (← .) を得て、その代入 {*X/(1 2 3)}を答とする。

それに対して、宣言型モデルでは、query から作った節：

```
(answer *X) ← (append (1 2) (3) *X).
```

をappend プログラムに加えて得られる

```
(answer *X) ← (append (1 2) (3) *X).
(append () *y *y) ← .
(append (*car . *cdr) *y (*car . *app))
    ← (append *cdr *y *app).
```

を初期プログラムとして、健全かつ完全な unfold 変換を行なう。この場合は、常に answer 節の body の第一 atom を選択して unfold 変換を行ない続けると、answer 節は、

```
(answer *X)
    ← (append (1 2) (3) *X).
(answer (1 . *Y))
    ← (append (2) (3) *Y).
(answer (1 2 . *Z))
    ← (append () (3) *Z).
(answer (1 2 3)) ← .
```

と変化していくので、最終プログラムとして

```
(answer (1 2 3)) ← .
(append () *y *y) ← .
(append (*car . *cdr) *y (*car . *app))
    ← (append *cdr *y *app).
```

を得る。

4.3 証明 vs. プログラム変換

論理モデルでは、計算とは証明 (背理法) であるという考えが基本にある。しかし、宣言的計算モデルでは、証明ではなく プログラム変換 (とくに unfold 変換) を計算と考えている。その根拠を述べる。

まず、証明とプログラム変換はその基礎がかなり違うことに注意したい。証明の基礎をなすのは、論理的帰結 ($\models$) の関係である。すなわち、論理式 E_1 から 論理式 E_2 が証明できるとは、$E_1 \models E_2$ が成り立つことと同値でなければならない。しかるに論理プログラムの等価変換は、論理的帰結 ($\models$) の関係に基づいた等価性を保存する変換とはみなすことができないことが知られている [13]。E_1 を unfold 変換して E_2 を得たとき、E_1 と E_2 が 論理的に等価 (すなわち、$E_1 \models E_2$ かつ $E_2 \models E_1$) とはかぎらない。

つぎに、宣言型計算モデルの基礎として、証明よりもプログラム変換のほうが有用であることを述べる。宣言的プログラムの世界では、unfold 変換を繰り返すことによって、refutation (背理法による証明) が達成できる (その例はすでに前節で見た)。unfold 変換よりも refutation (証明) は粒が大きい。したがって refutation に対する健全性と完全性の定理よりも、1 回の unfold 変換ごとの健全性と完全性の定理のほうが有用である。それから refutation に対す

る健全性と完全性の定理を導くことができるし、他のプログラム変換と組み合わせた定理を作ることもできる。また、unfold 変換は、refutation のように特定の goal に到着しなくても利用できる。

5 Lisp の場合

ここでは pure Lisp について考える。pure Prolog の場合と比較することによって、オブジェクト空間の違いがプログラミング言語の基本的な違いになることを示す。

5.1 Lisp プログラムの変換

次のプログラムは、list の append を行なう Lisp プログラムである。

```
(defun append (x y)
  (cond ((null x) y)
        (t (cons (car x)
                 (append (cdr x) y)))))
```

これに自然に対応している宣言的プログラムは次のものが考えられる。このプログラムを P_{app} とする。

```
(append *x *y *z)
    ← (null *x),
       (= *z *y).
(append *x *y *z)
    ← (consp *x),
       (car *x *car),
       (cdr *x *cdr),
       (append *cdr *y *app),
       (cons *car *app *z).
```

ここで、append やnull 、cons 、car 、cdr などが、関数から述語に変化している。null や = の定義は、

```
(null ()) ← .
(= *x *x) ← .
```

とすればよい。問題は、cons やcar 、cdr 、consp の定義である。たとえば car を定義するには、

```
(car (1 . 2) 1) ← .
(car (a . 2) a) ← .
(car (5 . b) 5) ← .
       :
```

と列挙的に無限の節を書くしかない。それは、Lisp の世界が (*A . *R) のような、変数のはいったコンスを許していないからである。これは、cons や cdr 、consp についても同様である。紙面の都合上省略するが、それぞれが無限個の節で定義してあるものとする。

5.2 Lisp のオブジェクト空間

Lisp を宣言型モデルでとらえる場合にどのようなオブジェクト空間が必要になるかを考えよう。

Lisp 言語で使われる式は、次の <Lisp S 式> で定義される。これは、変数の入らない S 式である。

```
<Lisp S式>
      = <アトム>
        | (<Lisp S式> . <Lisp S式>)
```

Lisp を 宣言的プログラムとみなすための、オブジェクト空間 $\mathcal{A}_L$ は、Lisp S 式または変数からなる。すなわち、

$\mathcal{A}_L$ = <Lisp S 式> | <変数>

Lisp のオブジェクト空間に作用している specialization は変数を任意の Lisp S 式または変数で置き換える代入である。

$\mathcal{S}_L$ = { <変数>/<$\mathcal{A}_L$ の元 >, … }

代入がオブジェクトにどう作用するかは明らかである。その写像を $\mu_L : \mathcal{S}_L \rightarrow map(\mathcal{A}_L)$ とするとき、$\Gamma_L =< \mathcal{A}_L, \mathcal{S}_L, \mu_L >$ が Lisp を宣言的プログラムとみなす場合のオブジェクト空間となる。

5.3 Lisp と Prolog のためのオブジェクト空間の比較

pure Lisp のためのオブジェクト空間：

$\Gamma_L =< \mathcal{A}_L, \mathcal{S}_L, \mu_L >$

と pure Prolog のためのオブジェクト空間：

$\Gamma_P =< \mathcal{A}_P, \mathcal{S}_P, \mu_P >$

を比較すれば、あきらかに次の包含関係がなりたつ。

$\mathcal{A}_L \subset \mathcal{A}_P$

$\mathcal{S}_L \subset \mathcal{S}_P$

$\mu_L \subset \mu_P$

ただし最後の式では、写像を集合と見ている。

$\mu_L = \{(s, a, b) \mid \mu_L(s)(a) = b\}$

$\mu_P = \{(s, a, b) \mid \mu_P(s)(a) = b\}$

Lisp のためのオブジェクト空間は Prolog のためのオブジェクト空間の真部分空間になっている。

5.4 式の評価に対応する unfold 変換

Lisp プログラムの append に対して、

```
(append (1 2) (3))
```

を与えると、評価結果として(1 2 3) が帰ってくる。

同じことを、対応する宣言的プログラムで行なうには、

```
(answer *X)
    ← (append (1 2) (3) *X).
(append *x *y *z)
    ← (null *x),
       (= *z *y).
(append *x *y *z)
    ← (consp *x),
       (car *x *car),
       (cdr *x *cdr),
       (append *cdr *y *app),
       (cons *car *app *z).
```

を初期プログラムとして、健全かつ完全な unfold 変換をanswer 節の body の先頭の atom に対して行ない続ける。まず、第1回目の unfold 変換では、answer 節は、

```
(answer *X)
    ← (null (1 2)),
       (= *X (3)).
(answer *X)
    ← (consp (1 2)),
       (car (1 2) *car),
       (cdr (1 2) *cdr),
       (append *cdr (3) *app),
       (cons *car *app *X).
```

の2つの節に展開される。しかしこの第1節は、さらなる unfold 変換で消滅する。第2節は繰り返し unfold 変換を受け、最終プログラムとして

```
(answer (1 2 3)) ← .
(append () *y *y) ← .
(append (*car . *cdr) *y (*car . *app))
    ← (append *cdr *y *app).
```

が得られる。

このときのプログラムの変化の様子は、もとの Lisp プログラムで評価が行なわれる計算とちょうど対応している。このような query に答える計算は、途中で得られる body が list の長さに応じて長くなり、スタックを使った lisp での append の計算に対応している。

5.5 unfold 変換

P_{app} は Γ_L 上のプログラムである。これをnull と = に関して unfold 変換すれば、

```
(append () *y *y) ← .
(append *x *y *z)
    ← (consp *x),
       (car *x *car),
       (cdr *x *cdr),
       (append *cdr *y *app),
       (cons *car *app *z).
```

を得る。cons 、car 、cdr 、consp などは無限の節をもつので、このプログラムをさらに unfold 変換すれば、結果として append の節が無限にできて得策ではない。したがって、これが Lisp の世界で unfold 変換したときの最良の結果であるといえる。

一方、P_{app} は Γ_P 上のプログラムと考えることもできる。Γ_P では、cons 、car 、cdr 、consp などは次のようにそれぞれ1個の節で定義できる。

```
(cons *car *cdr (*car . *cdr)) ← .
(car (*car . *cdr) *car) ← .
(cdr (*car . *cdr) *cdr) ← .
(consp (*car . *cdr)) ← .
```

これを用いて Γ_P 上で P_{app} を unfold 変換すれば、通常の Prolog で用いられている append の定義：

```
(append () *y *y) ← .
(append (*car . *cdr) *y (*car . *app))
    ← (append *cdr *y *app).
```

を得る。これを用いて(append (1 2) (3) *A) などの query に答える計算は、途中で得られる body が常に1つの atom からなるので、スタックを消費しない計算に対応している。

6 その他の場合

6.1 項書換えシステム

たとえば, 項書換えシステム

```
append(nil,X)  →  X
append(cons(Z,X),Y)
              →  cons(Z,append(X,Y))
```

に対応する宣言的プログラムは、次のようになる.

```
path(A,C) ← next(A,B), path(B,C).
path(A,A) ← .
next([append(nil,X), R],
     [X, R]) ← .
next([append(cons(Z,X),Y), R],
     [cons(Z,append(X,Y)), R]) ← .
```

項書換えシステムに対応するオブジェクト空間に特徴的な specialization は 2 種類ある。1 つは、項オブジェクトに対して代入をほどこすものであり、もう 1 つは、文脈を付け加えるものである。

6.2 制約論理プログラム

制約論理プログラムの節

```
p(Y) ← Y<3 | r(Y)
```

に対応する宣言的プログラムの節は、変数 P,R を用いて、

```
[P,{ P=p(Y),R=r(Y),Y<3 }]
    ← [R,{ P=p(Y),R=r(Y),Y<3 }]
```

とすればよい。制約論理プログラムに対するオブジェクト空間に特徴的な specialization はやはり 2 種類ある。1 つは、atom に対して代入をほどこすものであり、もう 1 つは、制約を追加するものである。

7 むすび

宣言型計算モデルという新しい計算モデルを提案した。そこでは、「計算とは宣言型プログラムのプログラム変換である」と考える。数多くのプログラミング言語や知識表現系における計算が宣言的計算モデルを基礎として統一的に議論できることがわかりはじめている。すでに述べたもの以外では、オブジェクト指向モデルや並列論理型言語などの「宣言的な部分」も宣言型計算モデルで議論できると考えられる。

プログラム変換、抽象解釈、タイプ推論などに宣言型計算モデルを適用する試みもはじまっており、よい感触を得ている。宣言型言語のプログラム変換を基礎として、自然言語の意味を計算したり、帰納的学習や演繹的学習などへの応用もなされている。宣言型計算モデルを基礎としたプログラミング言語として UL/α という言語 [9] がすでに利用されている。

宣言型計算モデルは、今後なすべきことはあまりにも多いが、これまでの計算モデルが生み出した多くの研究成果を自然に継承し、さらに発展させるために有効であると考えられる。宣言的プログラムの理論を基礎にすれば、いままでよりも複雑なあるいは高度なデータ構造を対象とした高次の計算を理論的な裏付けをもって達成できる。これは、理論的にも、実用的にも、意義があると考えられる。

参考文献

[1] Akama,K. : Declarative Semantics of Logic Programs on Parameterized Representation Systems. *Hokkaido University Information Engineering Technical Report*, HIER-LI-9001 (1990)

[2] Akama,K. : Generalized Logic Programs on Specialization Systems, *Preprints Work. Gr.for Programming. IPSJ* 6-4-PRG, pp.31-40 (1992)

[3] 赤間 清 : 一般化論理プログラムの Unfold 変換, 関数プログラミング JSSST'91, 近代科学社 (1992)

[4] Akama,K. : Sound and Complete Unifier Sets and mgu, *Hokkaido University Information Engineering Technical Report*, HIER-LI-9212 (1992)

[5] Akama, K.:A New Theoretical Foundation of Constraint Logic Programs, *Hokkaido University Information Engineering Technical Report*, HIER-LI-9220 (1992)

[6] 赤間清 : 項書換えシステムの論理的再構築, 日本ソフトウェア科学会, 関数プログランミング研究会, p.11 (1993)

[7] Apt,K.R. and van Emden,M.H. : *Contribution to The Theory of Logic Programming, J.ACM*, Vol.29, No.3 (1982)

[8] Clark,K.L. : Logic Programming Schemes, *Proceedings of the international conference on fifth generation computer systems* (1988)

[9] 出葉 義治, 繁田 良則, 赤間 清, 宮本 衛市 : 一般化論理プログラミング言語 UL/α のコンパイラ, 情報処理学会, プログラミング研究会, 9-7-PRG, pp.49-56 (1992)

[10] 井田哲雄 : 計算モデルの基礎理論, 岩波講座ソフトウエア科学 1 2, 岩波書店

[11] Jaffar,J. and Lassez,J.L. : Constraint Logic Programming, *Technical Report, Department of Computer Science, Monash University*, June (1986)

[12] 大松正樹, 赤間 清, 宮本 衛市 : 制約論理プログラムの部分計算, 日本ソフトウェア科学会, プログラム合成変換研究会, p.8 (1992)

[13] 玉木 久夫 : 論理型言語におけるプログラム変換, プログラム変換, 共立出版 (1987)

[14] 繁田良則, 赤間清, 宮本衛市 : 一般化論理プログラムとしての項書き換え系, 日本ソフトウェア科学会, 関数プログランミング研究会, p.10 (1992)