環境λ計算と単一化機構

西崎真也
shinya@math.s.chiba-u.ac.jp
千葉大学理学部数学・情報数理学科
〒263 千葉市稲毛区弥生町1番33号

著者は、関数型プログラミング言語の理論的モデルであるλ計算を拡張し、環境がファーストクラスな対象として扱えるような体系「環境λ計算」を提唱し研究してきた。環境λ計算では、環境を代入として形式化し、代入を項としてオブジェクトレベルで扱うことにより、環境をファーストクラスな対象として形式化することを可能にしている。
本論文では、単一化問題を環境λ計算の構文に導入し、その評価を単一化とし、そして、その解である単一化代入をファーストクラスな環境とみなすことにより、環境λ計算への単一化機構の融合をおこなった。

Environment Lambda Calculus and Unification

Shin-ya Nishizaki
Department of Mathematics and Informatics, Chiba University.
1-33, Yayoi-Cho, Inage-Ku, Chiba 263 JAPAN.

We proposed and have been studying a computational system, environment lambda calculus, which is based on lambda calculus and where we can use the first-class environment facility. We make possible it by formalization of environments as substitutions and by treatment of these substitutions in the object level rather than in the meta level.
In this paper we try to merge unification mechanism into the environment lambda calculus: we introduce syntactic forms for unification problems, identify evaluation of these syntactic forms with their unification, and regard unifiers obtained by unification as first-class environments.

1 はじめに

1.1 ファーストクラスな環境

環境は、プログラミング言語において基本的な概念の一つであり、計算の実行時における、変数とそれを束縛する値との対応である。一方、ファーストクラスなオブジェクトとは、数、リスト、文字列のように関数の引数や返値にすることができるデータのことである。例えば、整数はほとんどの言語でファーストクラスなオブジェクトである。しかし、手続きや関数は、必ずしもそうではない。例えば、言語 Pascal では、手続き名や関数名を引数として別の関数や手続きに渡すことはできるものの、返値として返すことはできない。それに対して、言語 Lisp をはじめとして関数型言語では、関数をファーストクラスなオブジェクトして扱えることが最大の特徴の一つである。また、Lisp の方言の Scheme では、関数のみならず継続もファーストクラスとして扱うことができ、その実現系の多くでは、環境もファーストクラスなオブジェクトとして扱うことができる。これらの Scheme の実現系において、ファーストクラスな環境の機能をサポートするプリミティブはいくつもあるが、その中で基本的なものとしては次の二つをあげることができる。

- `(the-environment)` このプリミティブが呼び出された時点における環境がデータとして返される。
- `(eval` *quoted-expression environment*`)` S-式の形で記述されている式 *quoted-expression* を環境 *environment* の下で評価した値が返される。

1.2 環境の代入による形式化とファーストクラスな環境

論文 [2, 3, 4] において、これまでファーストクラスな環境を λ 計算の枠組のもとで形式化することに取り組んできた。環境を形式化する手法として P.-L.Curien ら [1] により提唱された明示的代入 (explicit substitution) をもちいた。従来の λ 計算では、代入がメタな概念として扱われていたのであるのに対して、明示的代入のアイデアに基づく体系 $\lambda\sigma$ 計算では、代入の構文をあたえ、代入操作を体系の中で簡約規則としてあたえることにより、明示的に代入を扱う。これにより、環境が代入として形式化することが可能となる。

彼らの $\lambda\sigma$ 計算では、代入の構文クラスと項の構文クラスが明確に異なるものとして扱われているのだが、われわれは、この二つの構文クラスを一つにすることにより、ファーストクラスな環境が得られることを示した。このファーストクラスな環境を持つ λ 計算 λ_{env} の式は次のように与えられた。(詳細は [4] を参照)

$$\begin{aligned} e \quad ::= \quad & x \mid \lambda e.x \mid (ee') \\ & \mid id \mid (e/x)\cdot e' \mid (e \circ e') \end{aligned}$$

最初の 3 つの構文は通常の λ 計算の変数、ラムダ抽象、関数適用と同様である。後ろの 3 つのものがこの体系で新しく加わっているものである。id は恒等環境と呼ばれるもので、Scheme の `(the-environment)` に対応し、現在の環境を返す。$(e/x)\cdot e'$ は、環境拡張と呼ばれるもので、変数 x から式 e への束縛を環境 e' に追加したものを意味する。そして、$e \circ e'$ は環境合成と呼ばれるもので、式 e を環境 e' の下で評価した結果を返す。また、これらの新しいプリミティブに対して、従来の環境モデルは次のように拡張される。

$$\begin{aligned} [\![x]\!]\rho &= \rho(x) \\ [\![\lambda x.e]\!]\rho &= \lambda v.[\![e]\!](\rho[x \mapsto v]) \\ [\![ee']\!]\rho &= ([\![e]\!]\rho)([\![e']\!]\rho) \\ [\![id]\!]\rho &= \rho \\ [\![(e/x)\cdot e']\!]\rho &= ([\![e']\!]\rho)[x \mapsto [\![e]\!]\rho] \\ [\![e \circ e']\!]\rho &= [\![e]\!]([\![e']\!]\rho) \end{aligned}$$

この環境モデルからもわかることであるが、環境というものは、変数名から値への部分関数

(すなわち変数に対する代入) として把握されている。

1.3 単一化問題の意味としての代入

さて、環境の意味 (denotation) が代入であることは、以上の通りであるが、この他に、プログラミングの世界で、意味が代入であるものとしては、単一化問題が代表的なものである。単一化問題とは、第一階の項の等式の集合

$$\{t_1 = t_1' \cdots t_n = t_n'\}$$

であり、単一化問題の解決とは、これらの等式を成り立たせる単一化子と呼ばれる代入 σ を見つけることである。言い換えると、上の等式の集合は代入を意味としてもつ。

これらのことから、単一化問題を構文と導入し、その意味を単一化子とすることにより、ファーストクラスな環境をもつ λ 計算 λ_{env} を拡張することが考えられる。本論文では、単一化による λ_{env} の拡張を試み、その問題点を議論する。

2 プログラミング言語 λ_{eu}

2.1 構文の定義

定義 1 式. あらかじめ、変数の可算集合Varと関数記号の可算集合Funがあたえられているとする。そしてさらに、関数記号の集合Funから自然数の集合 $\mathcal{N}$ へのアリティ関数 $\mathsf{ar}(-)$ が与えられているものとする。

このとき、式は次のように帰納的に定義される：

$$\begin{array}{rcll} e & ::= & x & (x \in \mathsf{Var}) \\ & | & f(e_1, \ldots, e_n) & (f \in \mathsf{Fun}, n = \mathsf{ar}(f)) \\ & | & \lambda x.e & \\ & | & (e\ e') & \\ & | & id & \\ & | & (e/x) \cdot e' & \\ & | & (e \circ e') & \\ & | & \{e_1 = e_1', \ldots, e_l = e_l'\} & (l \geq 0) \end{array}$$

但し、x は変数を表すメタ変数で、e, e', e_1, e_2 などは、式を表すメタ変数である。

$\lambda x.e$ を (ラムダ) 抽象と呼び、$(e\ e')$ を (関数) 適用と呼び、id を恒等環境と呼び、$(e/x) \cdot e'$ を環境拡張と呼び、$(e \circ e')$ を環境合成と呼び、$\{e_1 = e_1', \ldots, e_n = e_l'\}$ をユニフィカンドと呼び、式の順序対 $e_i = e_i'$ の列である。この順序対のことを等式と呼ぶこととする。

また、上のようにして定義される式の集合を Expr とする。

2.2 計算規則

本論文では、λ_{eu}の計算規則の定義として自然意味論を与える。自然意味論は G. Kahn により提唱された意味論であり、プログラミング言語の式に対して意味を与える意味関数を構文的に定義することにより、プログラミング言語に対して意味を与える。この意味関数は通常、帰納的可算であり、操作的意味論とみなすこともでき、容易に論理型言語や関数型言語の上に実現することができることが特徴的である。

まず、意味関数を定義する前に、準備として式の構文クラスの部分集合である値という構文クラスを導入する。値とは、これ以上計算することのできないような式である。そしてさらに、その値の部分クラスである、一階項という構文クラスを導入する。本論文で扱う単一化は一階単一化である。一階項は単一化の対象となる。

定義 2 値. 式の集合Exprの部分集合であるValを以下のように帰納的に定義する。また、この集合Valの要素を値と呼び、メタ変数として、$v, v', v_1, v_2, \ldots$ などをつかう：

$$\begin{array}{rcl} v & ::= & x \mid x \circ w \mid f(v_1, \ldots, v_n) \mid uv \\ & & \mid (\lambda x.e) \circ v \mid id \mid (v/x) \cdot v' \end{array}$$

但し、w は、id および $(v/x) \cdot v'$ 以外の値を表すメタ変数であり、u は、$(\lambda x.e) \circ v$ 以外の値を

表すメタ変数であり、次のように定義できる：

$$\begin{array}{rcl} w & ::= & x \mid x \circ w \mid f(v_1, \ldots, v_n) \mid uv \\ & & \mid (\lambda x.e) \circ v \\ u & ::= & x \mid x \circ w \mid f(v_1, \ldots, v_n) \mid uv \\ & & \mid id \mid (v/x) \cdot v' \end{array}$$

定義 3 一階項. 一階項 (first-order term) は、次のように帰納的に定義される。

$$s \ ::= \ x \mid f(s_1, \ldots, s_n)$$

また、メタ変数としては、s, s_1, s_2, t, t_1, t_2 などを使う。

つぎに、意味関数を定義する。意味関数は式に対して直接定義されるのではなく、まず、式と環境から値への部分関数が与えられる。これは、第一引数としてあたえられる式を第二引数としてあたえられる環境のもとで評価した結果を値として返す部分関数である。本論文では、値呼び評価 *(call-by-value evaluation)* として意味関数を与えたい。値呼び評価の特徴は、環境において変数に対して束縛されているものはすべて値であることである。次に与える意味関数の引数としてあたえられる環境は、値となっているが、これは、環境に束縛されるものが値であることに対応する。

定義 4 三項関係 $\mathsf{ev}(-,-) = (-)$. 式 e と値 v と値 v' の間の三項関係 $\mathsf{ev}(e, v) = v'$ は次のように帰納的に定義される：

$$\frac{}{\mathsf{ev}(x, id) = x}$$

$$\frac{}{\mathsf{ev}(x, (v/x) \cdot v') = v}$$

$$\frac{\mathsf{ev}(x, v') = v''}{\mathsf{ev}(x, (v/y) \cdot v') = v''}$$

$$\frac{v \neq id \quad v \neq (v_1/x) \cdot v_2}{\mathsf{ev}(x, v) = x \circ v}$$

$$\frac{\mathsf{ev}(e_i, v) = v_i \ \ (i = 1, \ldots, n)}{\mathsf{ev}(f(e_1, \ldots, e_n), v) = f(v_1, \ldots, v_n)}$$

$$\frac{}{\mathsf{ev}(\lambda x.e, v) = (\lambda x.e) \circ v}$$

$$\frac{\left\{\begin{array}{l} \mathsf{ev}(e_1, v) = (\lambda x.e_1') \circ v_1 \\ \mathsf{ev}(e_2, v) = v_2 \\ \mathsf{ev}(e_1', (v_2/x) \cdot v_1) = v' \end{array}\right.}{\mathsf{ev}(e_1 e_2, v) = v'}$$

$$\frac{\left\{\begin{array}{l} \mathsf{ev}(e_1, v) = v_1 \\ v_1 \neq (\lambda x.e_1') \circ v_1'' \\ \mathsf{ev}(e_2, v) = v_2 \end{array}\right.}{\mathsf{ev}(e_1 e_2, v) = v_1 v_2}$$

$$\frac{}{\mathsf{ev}(id, v) = v}$$

$$\frac{\mathsf{ev}(e_1, v) = v_1 \quad \mathsf{ev}(e_2, v) = v_2}{\mathsf{ev}((e_1/x) \cdot e_2, v) = (v_1/x) \cdot v_2}$$

$$\frac{\mathsf{ev}(e_2, v) = v_2 \quad \mathsf{ev}(e_1, v_2) = v_1}{\mathsf{ev}(e_1 \circ e_2, v) = v_1}$$

$$\frac{\left\{\begin{array}{l} \mathsf{ev}(e_i, v) = s_i \ \ \mathsf{ev}(e_i', v) = s_i' \ \ (i = 1, \ldots, n) \\ \mathsf{unif}(\{s_i = s_i'\}_{i=1}^n) = \{x_1 \mapsto t_1, \ldots, x_l \mapsto t_l\} \\ \mathsf{ev}(e, v) = v' \end{array}\right.}{\mathsf{ev}(\{e_i = e_i'\}_{i=1}^n \cdot e, v) = (t_1/x_1) \cdot \cdots \cdot (t_l/x_l) \cdot v'}$$

$$\frac{\left\{\begin{array}{l} \mathsf{ev}(e_i, v) = s_i \ \ \mathsf{ev}(e_i', v) = s_i' \ \ (i = 1, \ldots, n) \\ \mathsf{unif}(\{s_i = s_i'\}_{i=1}^n) = \mathsf{failure} \\ \mathsf{ev}(e, v) = v' \end{array}\right.}{\mathsf{ev}(\{e_i = e_i'\}_{i=1}^n \cdot e, v) = v'}$$

但し、unifは、通常の一階項に対する単一化手続きであり、次のように定義される。

定義5 単一化手続きunif. ユニフィカンドを入力とし、出力として代入、もしくは、エラーを返す手続きunifを次のように定義する。

$$
\begin{aligned}
&\mathsf{unif}(\{\}) = \{\ \}\\
&\mathsf{unif}(\{x = x, \overline{t_n = t'_n}\})\\
&\quad = \mathsf{unif}(\overline{\{t_n = t'_n\}})\\
&\mathsf{unif}(\{x = s, \overline{t_n = t'_n}\})\\
&\quad \texttt{if } x \in \mathsf{FreeVar}(s) \texttt{ then raise failure}\\
&= \quad \texttt{else let } U = \mathsf{unif}(\overline{\{[s/x]t_n = [s/x]t'_n\}})\\
&\qquad\qquad \texttt{in } \{(x \mapsto U(s))\} \cup U\\
&\mathsf{unif}(\{s = x, \overline{t_n = t'_n}\}) = \mathsf{unif}(\{x = s, \overline{t_n = t'_n}\})\\
&\mathsf{unif}(\{f(s^1_1, \ldots, s^1_l) = f(s^2_1, \ldots, s^2_l), \overline{t_n = t'_n}\})\\
&= \mathsf{unif}(\{\overline{s^1_l = s^2_l}, \overline{t_n = t'_n}\})\\
&\mathsf{unif}(\{f(s^1_1, \ldots, s^1_l) = g(s^2_1, \ldots, s^2_{l'}), \overline{t_n = t'_n}\})\\
&= \texttt{raise failure}
\end{aligned}
$$

$\overline{v_n = v'_n}$ という記法は、いわゆる「ベクトル記法」で $v_1 = v'_1, \ldots, v_n = v'_n$ を意味する。

定義6 意味関数eval. 式から値への部分関数evalは次のように定義される：

$\mathsf{eval}(e) = v$ が成り立つのは、$\mathsf{ev}(e, id) = v$ が成り立つとき、またそのときに限る。

関数evalを定義するには、三項関係evが2引数関数であることが必要となるが、これはevの定義に関する帰納法をもちいて証明すれば簡単である。

例1. ここで、単一化の機構を用いた式の例をあげる。

まず、*cons*をアリティが2の関数記号とし、*nil* をアリティが0の関数記号とし、これら2つの関数記号を用いてリストを表現することする。さらに、$0, 1, 2, 3, \ldots$ がアリティ0の関数記号として導入されているものとする。そうすると、空でないリストが適用されたとき、その2番目の要素を返し、それ以外のものが適用されたときには*nil*を返す関数は次のように書ける：

$$\lambda l.(hd2 \circ \{l = cons(hd1, cons(hd2, tl))\} \cdot (nil/hd2) \cdot id\ )$$

実際に、この関数にリスト

$$cons(1, cons(2, cons(3, nil)))$$

を適用すると、ユニフィカンド

$$\{l = cons(hd1, cons(hd2, tl))\} \cdot (nil/hd2) \cdot id$$

が評価されたときの値は

$$(1/hd1) \cdot (2/hd2) \cdot (cons(3, nil)/tl) \cdot (cons(1, cons(2, cons(3, nil)))/l) \cdot (nil/hd2) \cdot id$$

となる。この環境の下で、変数*hd2*は評価されるので、結果は2となる。

一方、空でないリストではないもの、例えば、整数などを適用した場合は、単一化は失敗し、上記のユニフィカンドの評価値は、

$$(nil/hd2) \cdot id$$

となり、結果は、*nil*となる。

3 おわりに

この章では、本研究における問題点や将来の課題について検討する。

3.1 評価戦略と単一化

本論文で与えた計算規則は値呼び評価戦略である。この言語の前身であるファーストクラスな環境をもつλ計算λ_{env}に対しては弱簡約規則が与えられており、それに比べてかなり評価の順序に関して制約をかしたものになっている。値呼び評価戦略にした理由は次の通りである：ユニフィカンドが単一化の手続きにおいて成功するのは、それに含まれる式がすべて第一階の項の形をしていることが必要条件である。そのために、ユニフィカンドが単一化の手続きに適用される時には、最大限、ユニフィカンドに含まれる式は評価されていなければならない。従って、その時点での環境は評価ずみの値のみが束縛されていて、できうるかぎり評価が進ん

でなければならないのである。前述の例でいえば、ユニフィカンド

$$\{l = cons(hd1, cons(hd2, tl))\} \cdot (nil/hd2) \cdot id$$

に含まれる変数lが評価され具体化されていなければ、これの評価結果は単に、

$$(cons(hd1, cons(hd2, tl))/l) \cdot (nil/hd2) \cdot id$$

となってしまう。

3.2 関連研究

単一化の機構が組み込まれた関数型言語として、佐藤らによるQUTEがある。QUTEでは並列評価の機能があり、並列に実行される評価のスレッドは単一化を通して相互に通信が可能になっていることが特徴である。このような単一化機能による評価スレッド間の通信機能は、λ_{eu}は全くサポートできていない。この点はλ_{eu}における今後の課題である。

謝辞

本研究に関して議論および助言を下さいました住商情報システム株式会社の松木雅司さまに感謝いたします。また、QUTEに関する有益な情報を下さいました千葉大学理学部の桜井貴文さまに感謝します。

参考文献

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1:375–416, October 1991.

[2] Shin-ya Nishizaki. ML with first-class environments and its type inference algorithm. In *Logic, Language and Computation*, pages 95–116, 1994. Springer LNCS.

[3] Shin-ya Nishizaki. Simply typed lambda calculus with first-class environments. *Publication of Research Institute for Mathematical Sciences Kyoto University*, 30(6):1055–1121, 1995.

[4] Shin-ya Nishizaki. Type inference for simply-typed environment calculus with shadowing. In *Proceedings of the Fuji International Workshop on Functional and Logic Programming*, 1995.