依存型意味論における型推論の定式化と実装

佐藤 未歩 （指導教員：戸次 大介）

1 はじめに：依存型意味論 (DTS)

依存型意味論 (DTS)[1] は、自然言語の証明論的意味論の 1 つである。DTS では、文の意味をモデルを媒介として定義するのではなく、その文が正しいために何が必要か、そして、その文が正しいときに何が言えるかという、推論を媒介とした定義を与えている。結果的に、文の含意関係を、モデルを参照せずに直接計算することができる。それに加えて、照応解決および前提の束縛が proof search の問題に還元されるという特徴がある。

たとえば、*A man entered. He whistled.* という 2 文の意味表示はそれぞれ、図 1 と図 2 のようになる。2 文を組み合わせた意味表示は、図 3 のようになる。

$$\lambda c.\begin{bmatrix} u{:}\begin{bmatrix} x{:}\mathbf{entity} \\ \mathbf{man}(x) \end{bmatrix} \\ \mathbf{enter}(\pi_1(u)) \end{bmatrix}$$

図 1: A man entered.

$$\lambda c.\mathbf{whistle}(@_1 c)$$

図 2: He whistled.

$$\lambda c.\begin{bmatrix} v{:}\begin{bmatrix} u{:}\begin{bmatrix} x{:}\mathbf{entity} \\ \mathbf{man}(x) \end{bmatrix} \\ \mathbf{enter}(\pi_1(u)) \end{bmatrix} \\ \mathbf{whistle}(@_1(c,v)) \end{bmatrix}$$

図 3: A man entered. He whistled.

ここで、変数 c は local context と呼ばれ、先行する文の証明項が与えられる。ただし、先行する文が存在しない場合は、初期 context として () (unit) が与えられる。また、$@_1$ は underspecified term と呼ばれ、意味表示のうちで聞き手にとって未知の意味表示である。代名詞や前提トリガー（上の例では *He*）は、意味表示に必ず $@_i$ を含み、その内容を推定することが照応解決に対応する。DTS は型理論に基づいていることから、$@_i$ の型を推論することができる。$@_i$ はその型を持つ項によって置き換えられる。上の例では、$@_1$ は型

$$\begin{bmatrix} \top \\ \begin{bmatrix} u{:}\begin{bmatrix} x{:}\mathbf{entity} \\ \mathbf{man}(x) \end{bmatrix} \\ \mathbf{enter}(\pi_1(u)) \end{bmatrix} \end{bmatrix} \to \mathbf{entity}$$

を持つ。この型を持つ証明項には、$\lambda c.\pi_1\pi_2 c$ がある。これは 1 文目の entity を取り出すものであり、*a man* が先行詞であるという読みに対応している。

しかしながら、依存型理論において、一般に型推論は undecidable であるため、Agda[3] における型推論のように、構文的に制限された、依存型の部分体系を用いる必要がある。

本研究では、DTS のための型推論、型チェックアルゴリズムを定式化し、プログラミング言語 Haskell を用いて実装した。この体系は、Σ-type、$@_i$ を含むものであり、自然言語の意味表示を記述することができる。

2 DTS のための型チェック・型推論アルゴリズム

本研究において定義した体系は、Löh[3] の体系に基づき、DTS のための新たな構文をいくつか追加した体系となっている。この体系では、項は inferable term（型推論可能な項）と checkable term（型チェック可能な項）に分かれている。ただし、型推論可能ならば型チェック可能である。また、値は neutral term と value の 2 つに分かれている。

Löh[3] の体系では、application は inferable term であり、型推論が可能な項として定義されている。しかし、Löh[3] での application に対する規則である (ΠE) 規則だけでは、図 3 のようなケースにおいて $@_1$ の型を推論することはできない。図 3 の中には $@_1(c,v)$ という application が現れ、関数部分 $@_1$ は inferable ではなく、(ΠE) 規則では先に関数部分を評価するため、関数部分が inferable でない application に型をつけることはできないからである。一方で、application 全体の型が分かっており、かつ、引数部分の型が inferable である場合には、関数部分の型は定めることができるはずである。つまり関数部分の型は、引数部分の型を受け取り、application 全体の型を返すような関数型となると考えられる。以上のような推論を実現するためには、application を checkable term に追加し、引数部分を先に評価する (→E) 規則を追加する必要がある。

型推論、型チェックの各規則は図 4 のようになっている。これらの規則において、各規則の $\vdash$ の前後に現れている $[L]$ は、各 $@_i$ がどのような型を持つかを保持する環境を表しており、アスペランド環境と呼ばれる。照応解決などにおいて重要な $@_i$ は、$@_1, @_2, \ldots$ のように異なる pronoun の数だけ現れ、型推論・型チェック規則の実行に伴いアスペランド環境は更新されていく。実際にアスペランド環境に対する操作を行うのは型チェック規則の中の (ASP) 規則であり、この規則によって未知であった $@_i$ の型を定めることができる。

3 実装

DTS による自然言語の意味表示のための型推論アルゴリズムを、プログラミング言語 Haskell を用いて実装した。DTS における preterm（型のついていない項）は、Haskell 上で `Preterm` というデータ型を用意し、その型の値コンストラクタとして DTS の各構文を定義した。また、型推論は `typeInfer` という関数、型チェックは `typeCheck` という関数をそれぞれ定義した。

`typeInfer` と `typeCheck` には引数として 3 つのリストを渡しており、それぞれアスペランド環境、型環境、シグネチャを表現している。アスペランド環境は前節でも述べたように $@_i$ の型を保持する環境であり、

$$\frac{[L]\ \Gamma \vdash_\sigma M :\uparrow v\ [L']}{[L]\ \Gamma \vdash_\sigma M :\downarrow v\ [L']}\ (CHK)$$

$$\frac{[L]\ \Gamma \vdash_\sigma M :\uparrow (x{:}v) \to v'\ [L'] \quad [L']\ \Gamma \vdash_\sigma N :\downarrow v\ [L''] \quad v'[N/x] \twoheadrightarrow_\beta v''}{[L]\ \Gamma \vdash_\sigma MN :\uparrow v''\ [L'']}\ (\Pi E) \qquad \frac{[L]\ \Gamma \vdash_\sigma N :\uparrow v\ [L'] \quad [L']\ \Gamma \vdash_\sigma M :\downarrow v \to v'\ [L'']}{[L]\ \Gamma \vdash_\sigma MN :\downarrow v'\ [L'']}\ (\to E)$$

$$\frac{[L]\ \Gamma \vdash_\sigma M :\downarrow v\ [L'] \quad v'[M/x] \twoheadrightarrow_\beta v'' \quad [L']\ \Gamma \vdash_\sigma N :\downarrow v''\ [L'']}{[L]\ \Gamma \vdash_\sigma (M,N) :\downarrow \begin{bmatrix} x{:}v \\ v' \end{bmatrix}\ [L'']}\ (\Sigma I) \qquad \frac{[L]\ \Gamma \vdash_\sigma M :\uparrow \begin{bmatrix} x{:}v \\ v' \end{bmatrix}\ [L']}{[L]\ \Gamma \vdash_\sigma \pi_1 M :\uparrow v\ [L']}\ (\Sigma E)$$

$$\frac{[L]\ \Gamma \vdash_\sigma M :\uparrow \begin{bmatrix} x{:}v \\ v' \end{bmatrix}\ [L'] \quad v'[\pi_1 M/x] \twoheadrightarrow_\beta v''}{[L]\ \Gamma \vdash_\sigma \pi_2 M :\uparrow v''\ [L']}\ (\Sigma E) \qquad \frac{(i:\_) \notin L}{[L]\ \Gamma \vdash_\sigma @_i :\downarrow v\ [L,(i:v)]}\ (ASP) \qquad \frac{(i:v) \in L}{[L]\ \Gamma \vdash_\sigma @_i :\downarrow v\ [L]}\ (ASP)$$

図 4: 型推論・型チェック規則 (抜粋)

```
sig :: [(String, Preterm)]
sig = [("whistle", Imp (Con "entity") Type), ("entity", Type), ("man", Imp (Con "entity") Type), ("enter", Imp (Con "entity") Type)]

preTerm1 :: Preterm
preTerm1 = Sigma "v" (Sigma "u" (Sigma "x" (Con "entity") (App (Con "man") (Var "x"))) (App (Con "enter") (Proj One (Var "u"))))
(App (Con "whistle") (App (Asp 1) (Pair Unit (Var "v"))))

typeTest :: Maybe [(Int, Preterm)]
typeTest = typeCheck [] [] sig preTerm1 Type
```

図 5: テストプログラム

```
ghci> typeTest
Just [(1,Imp (Conj Top (Sigma "u" (Sigma "x" (Con "entity") (App (Con "man") (Var "x"))) (App (Con "enter") (Proj One (Var "u")))))
(Con "entity"))]
```

図 6: テスト結果

$@_i$ の番号 i (Int) と対応する型 (Preterm) のペアのリストである。型環境は変数の型を保持する環境、シグネチャは定数の型を保持する環境であり、どちらも変数名 (String) と対応する型 (Preterm) のペアのリストである。

型推論を実行する関数 typeInfer では、3 つの環境と型推論を行う項を受け取り、型推論の結果として返ってきた型と、更新されたアスペランド環境のペアを返している。その際型推論が失敗することもありうるので、失敗つき計算を表現するために Maybe モナドを用いている。型チェックを実行する関数 typeCheck では、3 つの環境と項と型を受け取り、与えられた項が与えられた型を持つかを確かめ、与えられた型を持つ場合のみアスペランド環境を返している。typeCheck でも typeInfer と同様の理由から Maybe モナドを用いており、型推論・型チェックに失敗した場合はどちらも Nothing が返る。

application に対する型チェック規則である (→E) 規則では、application の関数部分の型は、dependent functional type ではなく、依存関係のない implication となっている。よって (→E) 規則の実装では、application の関数部分の型が inferable であるかどうかを確かめる必要があり、inferable であったら (ΠE) 規則、inferable でなかったら (→E) 規則を適用すればよい。この考えに基づき、(→E) 規則は、まず (ΠE) 規則を実行し、失敗した場合には (→E) 規則を実行するよう実装を行っている。これにより、$@_i$ を含む application の型推論・型チェックが可能になる。

このアルゴリズムのテストとして、図 5 のようなテストプログラムを作成した。このプログラムは、第 1 節で述べた *A man entered. He whistled.* という文に対応する意味表示が型 type を持つという条件で型チェックを実行することにより、代名詞 *he* の意味表示中の $@_1$ への型割当を含むアスペランド環境を返す。このプログラムを実行することによって、$@_1$ の型を自動的に推論することができる。実行結果は図 6 のようになっており、これは理論的予測と一致する。

4 おわりに

本研究では、依存型意味論 (DTS) の部分体系に対する型チェック、型推論のアルゴリズムを提示し、実装した。この体系は部分体系ではあるものの、依存型 Π、Σ、underspecified term $@_i$ を含むもので、自然言語の意味論を展開するために十分な記述力を備えている。DTS では照応・前提の他にも、慣習的含みなど幅広い言語現象が $@_i$ を用いて説明されており [4]、本研究のアルゴリズムの適用が期待される。また、自然数型や等号型を含む、より豊かな体系に本研究のアルゴリズムを拡張することが考えられるが、今後の課題としたい。

参考文献

[1] Bekki, Daisuke. 2014. Representing Anaphora with Dependent Types. In Logical Aspects of Computational Linguistics (8th international conference, LACL2014, Toulouse, France, June 2014 Proceedings), N.Asher and S.Soloviev (Eds), LNCS 8535, pp.14-29, Springer, Heiderburg.

[2] Martin-Löf, Per. 1984. Intuitionistic Type Theory. vol. 17. Naples: Italy: Bibliopolis.

[3] Löh, A., C.McBride, and W.Swierstra. 2010. A Tutorial Implementation of a Dependently Typed Lambda Calculus. Fundamenta Informaticae - Dependently Typed Programming, Vol. 102, No. 2, pp. 177-207.

[4] Bekki, Daisuke and Eric McCready. 2014. CI via DTS. In Proceedings of LENLS11, pp.110-123.