依存型意味論における型推論の定式化と実装

佐藤 未歩 [1]　　戸次 大介 [2,3,4]

[1] お茶の水女子大学理学部情報科学科
g1120522@is.ocha.ac.jp
[2] お茶の水女子大学大学院人間文化創成科学研究科
bekki@is.ocha.ac.jp
[3] 国立情報学研究所
[4] 独立行政法人科学技術振興機構, CREST

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。形式意味論においては、モンタギュー意味論 (MG) や談話表示理論 (DRT) のようなモデル理論的意味論が主流である。DTS では、文の意味をモデルを媒介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 1 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

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

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。

一方、*he* の先行詞は *a man* ではなく、local context の中に存在する場合も考えられるが、その場合には別の証明項が与えられる。このように先行詞の曖昧性は、ある型に対して複数の異なる証明項が存在することに対応している。

DTS のこのような形式化は、DRT 等のモデル理論的な談話理論に対し、次のような利点を備えている。

1. 照応解決に推論が必要となる場合、たとえば bridgeing のようなケースは、厳密には DRT では扱うことができないが、DTS では proof search の枠組みの中で世界知識を用いることは自然である。
2. 照応や前提の先行詞が entity 以外であるケース、たとえば叙実動詞の前提のように先行詞が命題の形をとるような場合なども同様に扱える。
3. DRT における推論は、様々なシステムが提案されているが、いずれも first-order fragment に限られる。しかし、DTS では most などの GQ を含む higher-order ケースについても推論を行うことができる。

以上の議論は、DTS が基づいている Dependent Type Theory[2] に underspecified term を加えた体系において、型推論が可能であるという前提に基づいている。しかしながら、依存型理論において、一般に型

v	::=	n	neutral term
	|	type	the type of types
	|	kind	the type of type
	|	$\top$	top
	|	$\bot$	bottom
	|	$()$	unit
	|	$(x{:}v) \to v'$	dependent functional type
	|	$\begin{bmatrix} x{:}v \\ v' \end{bmatrix}$	dependent sum type
	|	$\lambda x.v$	lambda abstraction
	|	(v, v')	pair
n	::=	x	variables
	|	c	constants
	|	$@_i$	underspecified terms
	|	nv	functional application
	|	$\pi_i n$	projection

図 4: neutral term, value の定義

$M_\uparrow$	::=	x	variable
	|	c	constant symbol
	|	type	the type of types
	|	$(x{:}M_\downarrow) \to M_\downarrow$	dependent functional type
	|	$M_\uparrow M_\downarrow$	functional application
	|	$\begin{bmatrix} x{:}M_\downarrow \\ M_\downarrow \end{bmatrix}$	dependent sum type
	|	$(M_\uparrow, M_\uparrow)$	pair
	|	$\pi_i M_\uparrow$	projections
	|	$M_\downarrow : M_\downarrow$	annotated term
	|	$()$	unit
	|	$\top$	top type
	|	$\bot$	bottom type
$M_\downarrow$	::=	$M_\uparrow$	inferable terms
	|	$\lambda x.M_\downarrow$	lambda abstraction
	|	$(M_\downarrow, M_\downarrow)$	pair
	|	$@_i$	underspecified term

図 5: inferable term, checkable term の定義

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 自然言語における型推論の問題点

Löh[3] による Agda の型推論の体系においては、アノテーション構文を用いて、型推論が及ばない部分式の型をあらかじめ指定することで、一部の構文に対して型チェック、さらにその一部の構文に対して型推論を decidable に行うことを可能にしている。

しかし、Löh[3] の体系には Σ-type や $@_i$ が存在しないため、型規則を拡張する必要がある。さらに、自然言語の意味表示においては、Löh[3] の (IIE) 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ができない。なぜならば、図 3 において $@_1$ の型が推論できるという直感は、以下の (1)~(6) の推論に基づいているが、「(2),(5) より」という部分は Löh[3] の (IIE) 規則では実現できないからである。

(1) whistle の型は $\mathbf{entity} \to \mathsf{type}$ であること

(2) したがって、$@_1((), v)$ の型は $\mathbf{entity}$ であること

(3) $()$ の型は $\top$ であること

(4) v の型は $\begin{bmatrix} u{:}\begin{bmatrix} x{:}\mathbf{entity} \\ \mathbf{man}(x) \end{bmatrix} \\ \mathbf{enter}(\pi_1(u)) \end{bmatrix}$ であること

(5) したがって、$((), v)$ の型は $\begin{bmatrix} \top \\ \begin{bmatrix} u{:}\begin{bmatrix} x{:}\mathbf{entity} \\ \mathbf{man}(x) \end{bmatrix} \\ \mathbf{enter}(\pi_1(u)) \end{bmatrix} \end{bmatrix}$ であること

(6) (2),(5) より、$@_1$ の型は $\begin{bmatrix} \top \\ \begin{bmatrix} u{:}\begin{bmatrix} x{:}\mathbf{entity} \\ \mathbf{man}(x) \end{bmatrix} \\ \mathbf{enter}(\pi_1(u)) \end{bmatrix} \end{bmatrix} \to \mathbf{entity}$ である

これは、(IIE) 規則では application の関数部分を最初に評価するため、関数部分である $@_1$ の型が判明しなければ、その先の推論は行わないようになっているからである。このように、Löh[3] の体系を拡張し、(1)~(6) の推論を実現する上では、いくつかの技術的課題が存在している。

3 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。inferable term と checkable term、neutral term、value の詳細は図 4, 図 5 のように定義されている。本研究では inferable term に定数 c、type、dependent sum type、projection、pair、

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

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

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

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

図 6: 型推論・型チェック規則 (抜粋)

unit、top、bottom を新たに追加し、checkable term に pair、underspecified term、application を追加した。

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、先述したように、Löh[3] での application に対する規則である (IIE) 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、(IIE) 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。このとき、application は checkable term と inferable term の両方に存在することになる。

型推論、型チェックの各規則は図 6 のようになっている。これらの規則において、各規則のト記号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) 規則である。この規則が呼び出された際には、引数として呼び出された $@_i$ に対する型割当がアスペランド環境の中に入っている場合はアスペランド環境を更新せず、入っていない場合には型チェックの際に呼び出された型をその $@_i$ に割り当てるようにアスペランド環境を更新する。この操作を行うことによって、未知であった $@_i$ の型を定めることができる。

4 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ており、その型は図 7 のようになっている。

`typeInfer` と `typeCheck` の両方に引数として渡している 3 つのリストは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、$@_i$ 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であり、変数名 (`String`) と対応する型 (`Preterm`) のペアのリストである。シグネチャは定数の型を保持する環境であり、定数名 (`String`) と対応する型 (`Preterm`) のペアのリストである。これら 3 つの環境の中で、アスペランド環境と型環境は型チェック・型推論の実行時に更新されていくが、シグネチャは実行の前に与えておくことが前提となっており、基本的に更新されることはない。

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

```
typeCheck :: [(Int, Preterm)] -> [(String, Preterm)] -> [(String, Preterm)] -> Preterm -> Preterm -> Maybe [(Int, Preterm)]

typeInfer :: [(Int, Preterm)] -> [(String, Preterm)] -> [(String, Preterm)] -> Preterm -> Maybe (Preterm, [(Int, Preterm)])
```

図 7: 型推論・型チェックの型

```
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 8: 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 9: テスト結果

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、関数部分の型が、引数部分の型から結果の型への implication と一致するかどうか、パターンマッチを用いて判定すればよいと考えられる。しかしそのような実装では、application の関数部分の型が dependent functional type になっている場合、型チェックに失敗してしまう。というのは、Löh[3] の体系では application に対する型チェック規則は (CHK) 規則のみであるので、application に対して型チェックを行った場合、実際には型推論、すなわち (ΠE) 規則が適用されている。しかし、(→E) 規則が追加された体系では、application に対する型チェックの際には (CHK) 規則は適用されず、(→E) 規則のみが適用される。このため、関数部分の型が dependent functional type である場合には前述のパターンマッチに合致せず、型チェックに失敗してしまう。この問題を解消するためには、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 8 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 9 のようになっており、これは理論的予測と一致する。

5 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.