依存型意味論の証明探索とその実装に向けて

大洞 日音 (指導教員:戸次大介)

1 はじめに : 依存型意味論 (DTS)

含意関係認識は意味処理のための基本タスクであり、与えられたテキスト T から仮説 H が推論されるかどうかを判定する課題である。含意関係認識の一つの方針として、元の文を論理式に変換して表現するという方法がある。依存型意味論 (Dependent Type Semantics / DTS)[1] は証明論的意味論の一つであり、前提現象や照応現象など自然言語において基本的な現象を統一的に扱うことができる。DTS は依存型理論に基づいており、関数型を一般化した Π 型と直積型を一般化した Σ 型をもつ。

Π 型では $(x : A) \to B$ の後件 B が、前件 A の要素 x に依存することが可能であり、Σ 型では $(x : A) \times B$ の第二要素 B が、第一要素 A の要素 x に依存することが可能である。

自然言語の文と意味表示の対応は以下のように表現される。

$$\left[\begin{array}{l} u : \left[\begin{array}{l} v : \left[\begin{array}{l} x : \mathbf{entity} \\ \mathbf{girl}(x) \end{array}\right] \\ \left[\begin{array}{l} y : \mathbf{entity} \\ \mathbf{thesis}(y) \end{array}\right] \end{array}\right] \\ \mathbf{write}(\pi_1\pi_1 u, \pi_1\pi_2 u) \end{array}\right]$$

図 1: A girl writes a thesis.

$$\left[\begin{array}{l} a : \mathbf{entity} \\ \mathbf{girl}(a) \end{array}\right]$$

図 2: There is a girl.

型理論では意味の基本単位として $\Gamma \vdash a : A$ のような型判定を用いる。$\Gamma \vdash a : A$ は文脈 Γ において項 a は型 A を持つという意味である。依存型理論では統語レベルで項と型が区別されない。導入則と除去則に加えて形成規則 (formation rule) を追加することで型の定義を行なっている。Σ と Π の形成規則を図 5 に示す。図中の s_1 や s_2 は型としての型である **type** と **kind** のどちらかであり、これらの型を持つことを証明できたもののみを型として扱う。

証明論的観点からは $\Gamma \vdash a : A$ は文脈 Γ において命題 A の証明は a であるという解釈をすることもできる。項は証明に、型は命題に対応す るという関係はカリー=ハワード同型対応といい、これにより項は証明項とも呼ばれ、証明図は型推論図に対応している。

$$\left[\begin{array}{l} t : \left[\begin{array}{l} u : \left[\begin{array}{l} v : \left[\begin{array}{l} x : \mathbf{entity} \\ \mathbf{girl}(x) \end{array}\right] \\ \left[\begin{array}{l} y : \mathbf{entity} \\ \mathbf{thesis}(y) \end{array}\right] \end{array}\right] \\ \mathbf{write}(\pi_1\pi_1 u, \pi_1\pi_2 u) \end{array}\right] \end{array}\right] \vdash ? : \left[\begin{array}{l} a : \mathbf{entity} \\ \mathbf{girl}(a) \end{array}\right]$$

図 3: 証明探索の例

図 3 について証明探索を行うことで、証明項として $(\pi_1\pi_1\pi_1 t, \pi_2\pi_1\pi_1 t)$ を得ることができるが、これは $\left[\begin{array}{l} a : \mathbf{entity} \\ \mathbf{girl}(a) \end{array}\right]$ の証明図をエンコードしたものである。このように証明探索を用いることで照応解決や文間の推論などが実現する。しかし一般に証明探索は undecidable であることが知られており実装方法は自明ではない。先行研究として佐藤 [6] や piwek[5] で依存型理論の部分体系についての証明探索アルゴリズムが提案されている。また、Isabelle/HOL や Coq をはじめとするや依存型理論を用いた証明支援系の研究は近年盛んに行われており [3, 2]、多くの実装が存在する。本研究では DTS の部分体系のための証明探索アルゴリズムを、プログラミング言語 Haskell を用いて実装した。

2 DTS のための証明探索アルゴリズム

佐藤 [6] では DTS を未指定項を含む体系 (UDTT) と含まない体系 (DTT) にわけてそれぞれについて操作を行うことで、依存型理論の限定された体系について型推論を自動的に行うアルゴリズムが実装されている。ただ、DTT における型推論についてどのようなクラスについて推論可能かは明らかにされていない。一方 Piwek [5] では未指定項を含まない体系についてより広いクラスの推論を行うアルゴリズムがプログラミング言語 prolog を用いて実装されている。本研究において使用した体系は [5] の体系に基づき構成している。[5] では証明探索の際 Arrowterm という記法を用いている。Arrowterm では $(x{:}\,t_1) \to (..((y{:}\,t_2) \to t_n)..)$ という形で再帰的に Π 型を繰り返す項を $[x : t_1, y : t_2] => t_n$ と表記する。[5] では入力された prolog の変数やアトムをそのまま DTS の変項や定数項に流用し Arrowterm に正規化しているが、この場合形式の制限がないため、不正な入力を事前に阻止することができない。Haskell で直和型として Arrowterm を定義することで型が合わなかったり引数の数が合わなかったりという不適切な入力をトラップする防御的プログラミングを行うことができる。

3 実装

```
forward_context :: AEnv -> [AJudgement]

typecheck :: [Arrowterm] -- ^ context
  -> Arrowterm -- ^ term
  -> Arrowterm -- ^ type
  -> Int -- ^ depth
  -> Either (String) ([AJudgement])

deduce :: [Arrowterm] -- ^ context
  -> Arrowterm -- ^  type
  -> Int -- ^ depth
  ->Either (String) ([AJudgement])
```

図 4: forward_context,typecheck,deduce の型宣言

DTS における Preterm(型の付いていない項) は佐藤 [6] で用いられたデータ型 `Preterm` を用いた。また、`Preterm` をラップする形で Arrowterm のためのデータ型を定義した。型判定として、型環境 (`[Arrowterm]`)、項 (`Arrowterm`)、型 (`Arrowterm`) の三つのフィールドを持つデータ型 `AJudgement` を定義した。型環境は変数の型を保持する環境を指す。型推論は前向き推論のための `forward_context`、後ろ向き推論のための `deduce` という関数、型チェックは `typecheck` という関数を

$$(\Pi\text{-form})\ \frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x : A.B) : s_2} \qquad (\Pi\text{-intro})\ \frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x : A.B) : s}{\Gamma \vdash (\lambda x : A.b) : (\Pi x : A.B)} \qquad (\Pi\text{-elim})\ \frac{\Gamma \vdash F : (\Pi x : A.B) \quad \Gamma \vdash a : A}{\Gamma \vdash F \cdot a : B[x := a]}$$

$$(\Sigma\text{-form})\ \frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Sigma x : A.B) : s_2} \qquad (\Sigma\text{-intro})\ \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B[x := a] \quad \Gamma \vdash (\Sigma x : A.B) : s}{\Gamma \vdash (a, b) : (\Sigma x : A.B)}$$

$(s_1, s_2) \in \{\text{type}, \text{kind}\}$ $\qquad$ $s \in \{\text{type}, \text{kind}\}$

図 5: 形成規則

図 6: 型推論規則 (抜粋)

```
is_there_a_girl :: [AJudgement]
is_there_a_girl =
  let words = [write 2,thesis 1,girl 0,entity]
      sentences = [a_girl_writes_a_thesis 3 2 1 0]
      prop = there_is_a_girl 4 3
  in prove sentences words prop
```

図 7: テストプログラム

```
*DTS.Alligator.Test> is_there_a_girl
[ u0:[ u1:type ] =>[ u2:[ u3:[ u4:u1 ] =>⊥ ] =>⊥ ] =>u1, u5:type, u6:[ u7:u5 ] =>type,
u8:[ u9:u5 ] =>type, u10:[ u11:u5 ] =>[ u12:u5 ] =>type, u13:(u14:(u15:(u16:u5 )× u6 u16 )
× (u17:u5 )× u8 u17 )× u10 π1(π1(u14)) π1(π2(u14)) ⊢ (π1(π1(π1(u13))),π2(π1(π1(u13)))) : (u
18:u5 )× u6 u18]
```

図 8: テスト結果

それぞれ定義した。Haskell における型宣言を図 4 に示す。

前向き推論を実行する関数 `forward_context` では型環境を受け取り前向き推論の結果として得られた型判定のリストを返している。後ろ向き推論を実行する関数 `deduce` では型環境と型、深さを受け取り、6 に示す規則に従って操作を行い結果として得られた型判定のリストを返している。終わる確証のない再帰を繰り返すため、制御のために深さを用いた。深さ超過での計算中断などから型推論が失敗することもあるため Either モナドを用いている。[5] では 5 に示す形成規則も deduce 内で扱っていたが、形成規則を型推論で用いると無限に **type** 型の項を生成してしまう。Π-form を用いると $[x : A] \vdash A : \mathbf{type}$ と $[x : A, x : A] \vdash A : \mathbf{type}$ から $[x : A] \vdash (x{:}A) \to A : \mathbf{type}$ を得ることができるが、この得られた結果から $[x : A, x : A] \vdash A : \mathbf{type}$ と $[x : A] \vdash (x{:}A) \to A : \mathbf{type}$ から $[x : A] \vdash (x{:}A) \to (x{:}A) \to A : \mathbf{type}$ と再帰的に Π -form を用いることができてしまい、停止性が保証されない。このことから形成規則は型チェックを実行する関数 `typecheck` にわけて実装した。`typecheck` では型環境と項、型、深さを受け取り、5 に示す規則に従って操作を行い型チェックの結果として得られた型判定のリストを返している。`typecheck` でも `deduce` と同じ理由から Either モナドを用いており、計算中断時は `Left msg` という形でエラーメッセージが返る。

[5] での実装ともっとも異なるのは、変数の表現に de Bruijn index[4] を用いている点である。de Bruijn Index による実装は、表現がより緻密な整合性を求めるため、エラーが実装の早い段階で検出可能である。

de Bruijn index は、変数に言及する際に変数名でなく 0 以上の自然数で直接指定する記法である。自然数 k は「k 番目に束縛された変数」を表す。de Bruijn index を用いると $\left[\begin{smallmatrix} x:A \\ f(x) \end{smallmatrix}\right]$ という項は $\left[\begin{smallmatrix} A \\ f(0) \end{smallmatrix}\right]$ と書くことができる。ただ、de Bruijn index を用いる際は代入の際に変数の対応関係が崩れないように注意する必要がある。これについては代入時 [6] で用いられた関数 `ShiftIndices` を用いることで対応した。

このプログラムでは古典論理の体系 HK の基本的な定理 CM*や P についての証明を自動で行うことができる。7 では 図 3 について証明探索を行った。図 7 にプログラムを示す。簡単のため図 7 中では語彙と第一文を `words` と `sentences` に分けて定義したが証明探索時には `sentences` の後ろに `words` を結合し `context` として用いる。`a_girl_writes_a_thesis` をはじめとする要素は関数として別途定義しており、de bruijn index に基づき引数に該当する単語の `context` 上での位置に対応した実数を受け取り `Preterm` 型の項を返す。実行結果は図 8 に示しており、得られた証明項 $(\pi_1\pi_1\pi_1(u_{13}), \pi_2\pi_1\pi_1(u_{13}))$ 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) のうちの未指定項を含まない体系のための証明探索アルゴリズムについて考察し、その実装を行なった。この体系は部分体系であるものの、古典論理の体系 HK の基本的な定理について自動証明が可能であることを確認した。自然数型や統合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, D. and Mineshima, K.: Context-passing and Underspecification in Dependent Type Semantics, in *Modern Perspectives in Type Theoretical Semantics*, pp. 11--41, Springer (2017).

[2] Bertot, Y. and Castran, P.: Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions, Springer Publishing Company, Incorporated, æ1stÅ (2010).

[3] Nipkow, T., Wenzel, M. and Paulson, L. C.: *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, Springer-Verlag, Berlin, Heidelberg (2002).

[4] Pierce, B. C.: *Types and Programming Languages*, The MIT Press, 1st edition (2002).

[5] Piwek, P.: ALLIGATOR: THEOREM PROVING FOR DEPENDENT TYPE SYSTEMS WITH SIGMA TYPES (2006).

[6] 佐藤未歩 F 依存型意味論の証明探索とその実装, Master's thesis, お茶の水女子大学 大学院 (2016).