直観主義命題論理の自動定理証明器

tripplewhopper

2022 年 7 月 25 日

1. 要旨

Python3.8 を用いて直観主義下の命題論理の自動定理証明器を実装した．具体的には，与えられた命題をプログラムへの型付けとみなし，その型を与える単純型付きラムダ計算の項を出力ことで証明は完了する．実装上は型とラムダ計算の項はともに木構造で表した．適切な型を持つ項を探すには，バックトラッキングと列挙法を使った．

2. 約束

命題論理の式[1]とは，連言「∧」，条件法「→」，選言「∨」，否定「¬」という 4 つの命題結合子と，命題定項ボトム「⊥」で繋がった原子命題（アルファベット大文字）の組み合わせ（文字列）のことを指す．命題結合子の優先度は高い順から「¬」「∧」「∨」「→」とする．「¬」と「→」は右向き結合で「∧」と「∨」は左向き結合とする．

単純型付きラムダ計算とは，一般の型付きラムダ計算に組型と直和型を加えた拡張である．つまり

$$\tau^{2} ::= b|\tau_1 \to \tau_2|\tau_1 \land \tau_2|\tau_1 \lor \tau_2$$

$$b^{3} ::= \mathrm{P}|\mathrm{Q}|A|B| \perp \cdots$$

$$M^{4} ::= c^{5}|x|\lambda x{:}\tau.M|M_1\ M_2|(M_1, M_2)|\mathbf{fst}(M)|\mathbf{snd}(M)|\mathbf{inl}(M)|\mathbf{inr}(M)$$

および型付け規則 10 か条（Γは型環境）

$$\Gamma \vdash c^{\tau}{:}\tau \tag{1}$$

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x{:}\tau} \tag{2}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash M{:}\tau_2}{\Gamma \vdash \lambda x{:}\tau_1.M{:}\tau_1 \to \tau_2} \tag{3}$$

$$\frac{\Gamma \vdash M{:}\tau_1 \to \tau_2 \quad \Gamma \vdash N{:}\tau_1}{\Gamma \vdash M\ N{:}\tau_2} \tag{4}$$

$$\frac{\Gamma \vdash M{:}\tau_1 \quad \Gamma \vdash N{:}\tau_2}{\Gamma \vdash (M, N){:}\tau_1 \land \tau_2} \tag{5}$$

$$\frac{\Gamma \vdash M{:}\tau_1 \land \tau_2}{\Gamma \vdash \mathbf{fst}(M){:}\tau_1} \tag{6}$$

---

[1] 岡本 賢吾，*記号論理学教材*, “命題を代理させるために言語内に導入される記号表現，すなわち，先に「命題表現」と 呼んだものは，ふつう「整式 well-formed formula」，簡単には「式」と呼ばれる．”

[2] 型

[3] 基本型

[4] 項

[5] 定数

$$\frac{\Gamma \vdash M: \tau_1 \wedge \tau_2}{\Gamma \vdash \mathbf{snd}(M): \tau_2} \tag{7}$$

$$\frac{\Gamma \vdash M: \tau_1}{\Gamma \vdash \mathbf{inl}(M): \tau_1 \vee \tau_2} \tag{8}$$

$$\frac{\Gamma \vdash M: \tau_2}{\Gamma \vdash \mathbf{inr}(M): \tau_1 \vee \tau_2} \tag{9}$$

$$\frac{\Gamma \vdash L: \tau_1 \vee \tau_2 \quad \Gamma, x_1: \tau_1 \vdash M_1: \tau \quad \Gamma, x_2: \tau_2 \vdash M_2: \tau}{\Gamma \vdash \mathbf{case}\ L\ \mathbf{of}\ \mathbf{inl}(x_1) \Rightarrow M_1 \,|\, \mathbf{inr}(x_2) \Rightarrow M_2: \tau} \tag{10}$$

のこととする．通常，組型の型は$\tau_1 \times \tau_2$，直和型の型は$\tau_1 + \tau_2$と書かれているが，ここではカリー＝ハワード同型対応により，それぞれは連言と選言の式に相当するので，便宜上「∧」と「∨」で書いたのである．混同が起こらない限り，「型」と「命題表現」は区別しなくても良いであろう．

3. モジュール構造

本プログラムは三つのモジュール

```
lambda_interface.py,
simply_typed_lambda_calculi.py,
simply_typed_lambda_parser.py
```

からなっている．

`lambda_interface.py` には，一般的なラムダ計算の項を表す抽象クラス `ILambdaTerm` が定義されている．

`simply_typed_lambda_calculi.py` には，`ILambdaTerm` を継承した型付きラムダ計算の項を表す抽象基底クラス `ITypedLambda` と「型」を表すための抽象基底クラス `IType` が定義されている．

`simply_typed_lambda_parser.py` には，命題表現を再帰下降法で解析する関数

```
parse_type(s: str, typenames: List[str])
```

が実装されている．文法の詳細は付録を参照すること．

4. 入力の形式

Coq と似たように，処理の便宜上に

表 1

命題表現	入力
連言「$A \wedge B$」	`r`[6]`'A/\B'`
条件法「$A \rightarrow B$」	`'A->B'`
選言「$A \vee \mathrm{B}$」	`r'A\/B'`
否定「$\neg A$」	`'~A'`

とする．また，前提は全て条件法で与えられなければならない．例えば，

$$P, Q \vdash R$$

は’`P->Q->R`’と入力する.(’`Q->P->R`’でもよい)

[6] Python で raw 文字列を表す.

5. データ構造

5.1. `class IType` 型または命題

入力した命題表現, 例えば `r'(A->~A)->~(~A->A)'` に対して構文解析を行ったあと, `IType` オブジェクトが得られる。その `IType` オブジェクトは木構造をしており, 例の場合, 木構造のルートは(`IType` のサブクラスである)`Impl` 型のオブジェクト（Implication の略より）であり, 「$? \to ?$」という構造を表す[7].

ここで注目すべき点は, 否定「$\neg A$」を表示する `class Not` は $A \to \bot$ として処理するため `Impl` を継承している. `IndividualType` は, 原子命題 A, B, C ...を表示するため, 処理の便宜上に `str` のサブクラスとして定義した. ボトム「$\bot$」は `IndividualType` のサブクラス `Falsum` として定義した.

「$? \to ?$」「$? \wedge ?$」「$? \vee ?$」の構造は似ているため, 二項演算子を表す `BinaryTypeTerm` を作ってそのサブクラス `Impl`, `And`, `Or` として定義し, 命題結合子の左側の「?」は `self.t0` と、右側の「?」は `self.t1` と名付けた.

5.2. `class ITypedLambda` 単純型付きラムダ演算の項

`ITypedLambda` は 9 個のサブクラスを持っている.

表 2

サブクラス	対応する項
`class TypedVariable`	$x : \tau$
`class TypedFuncDef`	$\lambda x : \tau . M$
`class TypedApply`	$M\ N$
`class TypedPair`	(M, N)
`class TypedFirst`	$\mathbf{fst}(M)$
`class TypedSecond`	$\mathbf{snd}(M)$
`class TypedCaseOf`	$\mathbf{case}\ L\ \mathbf{of}\ \mathbf{inl}(x_1) \Rightarrow M_1 \mid \mathbf{inr}(x_2) \Rightarrow M_2$
`class TypedInl`	$\mathbf{inl}(M)$
`class TypedInr`	$\mathbf{inr}(M)$

`ITypedLambda` のサブクラスはインスタンス化するたびに項の型をチェックするため, インスタンス化が成功すれば合法的な型を持つ項であると言える. 項の型を取り出すにはメソッド

```
@abstractmethod
def get_type(self) -> IType:
    """
    This method gets the IType bound to a certain term.
    """
    ...
```

を使えばよい.

ほかにも `ForAllType`, `ExistsType` などがあるが, それは述語論理の定理証明（未完成）に使われる予定のものであり, 今は無視してよい.

6. 自動証明のアルゴリズム

6.1. ある型（命題, `IType` オブジェクト）を証明するには IType の再帰的メソッド

[7] ここで「?」は任意の `IType` オブジェクトを表す.

```
    @abstractmethod
    def deduce(self,
               env: List['ITypedLambda'],
               hypothesis: Dict['IType', 'ITypedLambda'],
               visit: Dict['IType', int])
               -> 'ITypedLambda':
        ...
```

を呼び出す必要がある. `IType` のサブクラスは, それぞれ自分独自の `deduce` メソッドを持っている. 引数は `env`, `hypothesis`, `visit` である.

6.1.1. `env`

もともと再帰の中でコンテキストにあるラムダ抽象で束縛された変数のリストとされてきたが, 今は基本的にはデバッグのために存在する（無限再帰に陥る場合は `env` が急速に長くなるので `assert len(env)<30` とチェックしておく）.

6.1.2. `hypothesis`

再帰の中で「今まで得られた単純型付きラムダ計算の項（つまり命題の証明）とその命題」を格納する辞書（いわゆる型環境Γ）である. 証明を試行錯誤するときはその中からいくつかの型を取り出し組み合わせを試し, 新たに矛盾のない型ができる際にその型と同じように項を組み合わせて, 辞書を更新する.

6.1.3. `visit`

無限再帰に陥るのを防ぐための辞書である. 辞書型を使ったのはコードを数度書き直したためであるが, 今は Set と全く同じように使っている.

6.1.4. 戻り値

型に対応する一つの証明, すなわち単純型付きラムダ計算の項である.

6.2. 次に, 各 IType のサブクラスの独自の `deduce` 実装を説明する.

6.2.1. `And.deduce`

「$A \wedge B$」を証明するにはAとBをそれぞれ証明[8]しなければならない. 共に結果が得られれば

$$(A\text{の証明}, B\text{の証明})$$

にして返す.

6.2.2. `Not.deduce`

「$\neg A$」を証明するには「$A \to \bot$」を証明すればよい.

6.2.3. `Or.deduce`

「$A \vee B$」を証明することは「Aを証明」または「Bを証明」することになるが, まず「Aの証明」を試し, 成功すれば

$$\mathbf{inl}\left(A\text{の証明}\right) : A \vee B$$

を返す. もし失敗なら次に「Bの証明」を試し, 成功すれば

$$\mathbf{inr}\left(\text{B の証明}\right) : A \vee B$$

[8] 「命題〇〇を証明する」＝「`IType` オブジェクト〇〇の `deduce` メソッドを呼び出す」

を返す．もし失敗なら 6.2.5(vii)(viii)と同じような手続きを実行してみる．それも失敗した場合, `DeductionFailed` という `RuntimeError` を継承した例外の送出を行う．

6.2.4. `Impl.deduce`

「$A \to B$」の証明はAによる場合分けが必要である．

6.2.4.1. Aが `IndividualType` または `Impl` の場合

型付き変数x[9]: Aを新たに作り, `env` の後ろに追加し, `hypothesis` をx: Aで更新する[10]．そしてBの証明を試す．成功するなら

$$\lambda x{:}\,A.\,B\text{の証明}: A \to B$$

を返す．

ただし, `env` の後ろ[11]からx: Aを削除することと `hypothesis` からx: Aが**自由変数**として出現する全ての項を削除することとを忘れなければならない．この削除操作は，証明が成功しても失敗しても行われなければならない．特に Python においては `try…except…finally` の `finally` 文を使う必要がある．

6.2.4.2. Aが `And` の場合

$A(= A' \wedge A''$とする$)$を型に持つ型付き変数x: $A' \wedge A''$を新たに作り，ただしx: $A' \wedge A''$ではなく

$$\mathbf{fst}(x): A', \mathbf{snd}(x): A''$$

を `env` に追加する。`hypothesis` を$\mathbf{fst}(x): A'$と$\mathbf{snd}(x): A''$で更新する．

次に, 6.2.4.1 の場合と同様にBの証明を試す．削除操作に関しては, `env` の後ろからから$\mathbf{fst}(t_i): A'$と$\mathbf{snd}(t_i): A''$をそれぞれ削除して，ただし `hypothesis` からは依然としてx: $A' \wedge A''$を**自由変数**とする全ての項を削除する（もちろん$\mathbf{fst}(x)$も$\mathbf{snd}(x)$も削除される）．

6.2.4.3. Aが `Or` の場合

$A(= A' \vee A''$とする$)$についてさらに場合分けが必要である. Coq では `destruct` という命令で場合分けができるが，ここで `Or` 型にも `Or.destruct` というメソッドを実装した．あとで説明するので，今は `Or.destruct` は

(i) A'を `hypothesis` に追加する場合のBの証明
(ii) A''を `hypothesis` に追加する場合のBの証明

と場合分けして試して
$A' \to B$の証明

$$\lambda y{:}\,A'.\,B\text{の証明}: A' \to B$$

と$A'' \to B$の証明

$$\lambda z{:}\,A''.\,B\text{の証明}: A'' \to B$$

---

[9]実装上は自動生成された型付き変数の名前がすべて異なるようにした

[10] `hypothesis` を型Aの項Mで更新することは `hypothesis.setdefault(`A`, `M`)`のことである．更新は必ずしも成功しない．それは，先に存在していた同じ型（ここではA）の項M'があるとすれば，M'に含まれる束縛変数のスコープは，Mの中の束縛変数のスコープより狭いことはないから，M'のままで有利，少なくとも損はないであるから．

[11] バックトラッキングの特徴から `env` の最後にx: Aが位置することは保証される．

を共に返すことだけ分かればよい（片方が失敗したとしても例外を送出する）．`Or.destruct` を呼び出したら型付き変数$x: A' \vee A''$を新たに作り，

$$\mathbf{case}\ x\ \mathbf{of}\ \mathbf{inl}(y) \Rightarrow \left(\left(\lambda y: A'.B\text{の証明}\right)\ y\right)$$

$$|\ \mathbf{inr}(z) \Rightarrow \left(\left(\lambda z: A''.B\text{の証明}\right)\ z\right)$$

$$: A \to B$$

を返す．

6.2.5. `IndividualType.deduce`

(i) `hypothesis` を走査し，型付け規則(4)を適用できる

$$M: \tau_1 \to \tau_2, N: \tau_1$$

を探し，新たな型ができるかどうかチェックする．できるなら `hypothesis` を$M\ N: \tau_2$で更新し，(i)を繰り返す．できないなら(ii)に移る．

(ii) [12]`hypothesis` を走査し，型付け規則(6)(7)を適用できる項Mを探し，$\mathbf{fst}(M)$と$\mathbf{snd}(M)$で `hypothesis` を更新する.(ii)を繰り返す．できないなら(iii)に移る．

(iii) `hypothesis` を走査し，$M: \tau_1 \to \tau_2$のような項で
  (a) τ_2が `hypothesis` にない
  (b) τ_1が `visit` にない
  (c) τ_1が `self`[13]に等しくない

  3 つの条件を満たすものを探す. (iv)に移る．

(iv) (iii)で見つけられた項を順に$M^{(i)}: \tau_1^{(i)} \to \tau_2^{(i)}\left(i = 1,2, \ldots, K_{\text{Impl}}\right)$とすると，$i = 1,2, \ldots K_{\text{Impl}}$に対して `visit` に$\tau_1^{(i)}$を追加する．$K_{\text{Impl}} = 0$の場合（見つからなかった）は(v)に移る．

(v) $i = 1,2, \ldots$に対して$\tau_1^{(i)}$の証明を求める．成功した場合，その証明を$N^{(i)}: \tau_1^{(i)}$とすると，$M^{(i)}\ N^{(i)}: \tau_2^{(i)}$で `hypothesis` を更新する．失敗した場合は `hypothesis` を更新せず，続いて$\tau_1^{(i+1)}, \tau_1^{(i+2)}, \ldots$の証明を求める. (vi)に移る．

(vi) 新たに（今回の繰り返しで）`hypothesis` に追加する項が 1 つでもあれば(i)に戻る．
  そうでないなら(vii)に移る．

(vii) `hypothesis` を走査し，$W: \mu_1 \vee \mu_2$のような項で，μ_1もμ_2も `hypothesis` にないようなものを探す. (viii)に移る．

(viii)(vii)で見つけられた項を順に$W^{(j)}: \mu_1^{(j)} \vee \mu_2^{(j)}(j = 1,2, \ldots, K_{\text{Or}})$とすると，$j = 1,2, \ldots K_{\text{Or}}$に対して以下の手続きを行う．
  (a) `visit` に `self` を追加する．

---

[12] 理論的には更新できなくなるまで(ii)を繰り返した方が良いが，一通りだけであっても無事に証明できるようである．

[13] `self` も型オブジェクトである．

(b) $\mu_1^{(j)} \vee \mu_2^{(j)}$の `destruct` メソッドを呼び出し，場合分けの `self` の証明

$$\lambda y{:}\,\mu_1^{(j)}.\,\mathrm{self}\ の証明{:}\,\mu_1^{(j)} \to \mathrm{self}$$

と

$$\lambda z{:}\,\mu_2^{(j)}.\,\mathrm{self}\ の証明{:}\,\mu_2^{(j)} \to \mathrm{self}$$

を求める．

(c) 呼び出しが成功した場合, `self` の証明は

$$\begin{aligned} \textbf{case}\ W^{(j)}\ \textbf{of inl}(y) &\Rightarrow \left(\left(\lambda y{:}\,\mu_1^{(j)}.\,\mathrm{self}\ の証明\right)\ y\right) \\ |\ \textbf{inr}(z) &\Rightarrow \left(\left(\lambda z{:}\,\mu_2^{(j)}.\,\mathrm{self}\ の証明\right)\ z\right) \\ &{:}\,\mu_1^{(j)} \vee \mu_2^{(j)} \to \mathrm{self} \end{aligned}$$

と得られる．これで `hypothesis` を更新し，戻り値として返す．
呼び出しが失敗した場合, `visit` から `self` を削除して，続いて$j+1, j+2, \ldots K_{\mathrm{Or}}$の場合を考慮する．

(ix) (viii)で新たに `hypothesis` に追加する項が 1 つでもあれば(i)に戻る．
そうでない場合は証明失敗として `DeductionFailed` を送出する．

6.3. `Or.destruct` に関する説明

```
def destruct(self,
             goal: IType,
             env: List['ITypedLambda'],
             hypothesis: Dict['IType', 'ITypedLambda'],
             visit: Dict['IType', int]) \
        -> 'Tuple[TypedFuncDef, TypedFuncDef]':
```

$\mathrm{self} = \pi_1 \vee \pi_2$のとする．
場合分けしてgoalの証明を求めるには
`Impl.deduce` を 2 回利用して， $\pi_1 \to \mathrm{goal}$の証明

$$\lambda w{:}\,\pi_1.\mathrm{goal}\ の証明{:}\,\pi_1 \to \mathrm{goal}$$

と$\pi_2 \to \mathrm{goal}$の証明

$$\lambda t{:}\,\pi_2.\mathrm{goal}\ の証明{:}\,\pi_2 \to \mathrm{goal}$$

が共に成功した場合に限りそれらを返す．

6.4. `hypothesis` に関する説明

新しい命題の証明ができる度に hypothesis を更新する．また，ある命題の証明を求めるときはまず hypothesis を引いてすでにある場合は直接それを返す．

6.5. `visit` の役立ち方

もし`hypothesis`に新しい型が追加されなければ，再帰の連続の中で同じ型の`deduce`メソッドを呼び出す必要がない．呼び出してしまっても`hypothesis`に何の変化をもたらさず，そのまま無限再帰に陥ることになる．それを防ぐために6.2.5(iii)(b)のように`visit`をチェックする．ここ1ヶ所で十分なのは無限再帰の場合は必ず`IndividualType.deduce`を呼び出すからである．

「もし`hypothesis`に新しい型が追加されなければ」というのは，「もし`hypothesis`に新しい型が追加されることができたら」以前見けられなかった証明は見つかるかもしれないことを意味している．したがって`hypothesis`への更新が成功すれば直ちに`visit`を空に[14]すべきである．これはあくまでヒューリスティック(heuristic)な方法[15]だとわかるが，意外に有効である．

7. 計算量に関する分析

選言命題に対してはバックトラッキングがあるため，最悪の場合は選言命題の数の指数時間を要することになる．空間的計算量は`deduce`メソッドの引数である`hypothesis`の大きさに依存する．hypothesisの大きさは上に有界なはずだが... $A -> A -> A$と左端に無限に伸ばすという型ができることがないのを証明できていない．もし証明できれば，命題を二分木で表示する観点で高さが有限な二分木はカタラン数ほどあるのでそれが上界の一つだと考える

8. 参考文献

https://coq.vercel.app
岡本 賢吾，*記号論理学教材 命題論理. NJ/NK2022版*
https://docs.python.org/3.8/library/index.html
https://dl1.cuni.cz/pluginfile.php/334385/mod_resource/content/1/Logic%20and%20Structure%20-%20CH5.pdf
http://strictlypositive.org/Easy.pdf
http://www.cse.chalmers.se/~bengt/papers/GKminiTT.pdf

9. 付録

自動証明器の単純型付けラムダ計算の文法[16]

ただし，太文字は非終端記号で，終端記号（文字列リテラル）はシングルクォーテーション「'」で囲まれる．イタリックの*identifier*終端記号は`Python`の`str.isidentifier`関数が`True`を返す文字列，いわゆる識別子である．

S → **Term** | **Term** ':' **Type**
Term → **Apply**
Apply → **Apply NonApp** | **NonApp**
NonApp → **FuncDef** | '(' **Term** ')' | **Pair** | **Variable** | **Inl** | **Inr** | **First** | **Second** | **CaseOf**
Inl → 'inl' '(' **Term** ')'
Inr → 'inr' '(' **Term** ')'

---

[14] `visit.clear()`を呼び出すこと．

[15] つまり`visit`が空でないことは「`hypothesis`の状態が変化していない」であるための十分条件であり，必要条件でない．

[16] https://ja.wikipedia.org/wiki/文脈自由文法

First → `'fst'` `'('` **Term** `')'`

Second → `'snd'` `'('` **Term** `')'`

CaseOf → `'case'` **Term** `'of'` **Inl** `'=>'` **Term** `'|'` **Inr** `'=>'` **Term**

Pair → `'('` **Term** `','` **Term** `')'`

FuncDef → `'λ'` **Variable** `':'` **Type** `'.'` **Term**

Variable → *identifier*

Type → **ImplType**

ImplType → **ImplType** `'->'` **SumType** | **SumType**

SumType → **SumType** `'\/'` **ProductType** | **ProductType**

ProductType → **ProductType** `'/\'` **NotType** | **NotType**

NotType → `'~'` **NotType** | **SingleType**

SingleType → **IndividualType** | `'('` **Type** `')'`

IndividualType → *identifier* | `'⊥'`