Jacques Garrigue, 2007 年 12 月 12 日

12 推論と単一化

帰結

前回は命題論理式の正規化を見た．選言標準形にすることで，元の式が充足可能かどうかが簡単にチェックできる．

$$P = \bigvee_i (\bigwedge_j a_{ij} \wedge \bigwedge_k \neg b_{ik})$$

全ての i について，ある j と k で $a_{ij} = b_{ik}$ ならば，P が真になれない．そうでなければ，a_{ij} と b_{ij} が全て異なる i を選んで，a_{ij} を真と仮定し，b_{ik} を偽と仮定すれば P が充足される．

では，二つの式 P と Q があったとき，$P \to Q$ が恒真かどうかをどう調べればよいか？ 一つの方法は $\neg P \vee Q$ が恒真かどうかを調べることだが，逆に $P \wedge \neg Q$ が充足不能であることを調べるという方法もある．特に Q が連言を含まない場合，P をあらかじめ選言標準形にして置けば，それぞれの連言式に $\neg Q$ を足すだけで新たな選言標準形が得られ，効率よく充足可能性を調べられる．

```
val to_dnf : prop -> (string list * string list) list

let contradiction : (string list * string list) list -> bool =
  List.exists (fun (lt, lf) -> List.exists (fun a -> List.mem a lf) lt)

let implies p (q : string) =
  contradiction (List.map (fun (lt, lf) -> (lt, q :: lf)) p)

# implies [["a"],["b"]; ["b";"a"],["c"]] "a";;
- : bool = true
```

元々真と仮定されている情報に確かめたい帰結の否定を追加することで矛盾を作る方法を反駁という．論理型プログラミング言語の基礎になっている．

ホーン節

上に見たように，選言標準形は帰結を調べるのに便利だが，実際にはあまり使えない．なぜならば，知識は貯まるものであり，新しい知識の追加は自然に連言という形で表現される．そうすると，選言標準形がどんどん膨れていく．

連言標準形でありながら，帰結を調べるのに適している形が知られている．ただし，連言標準形の中の各選言部分に対して条件がある．肯定が一つだけと決められている．

$$\bigwedge_i (a_i \vee \neg b_{i1} \vee \ldots \vee \neg b_{ik})$$

ならばの定義を思い出すとこれは以下のように書ける

$$\bigwedge_i ((b_{i1} \wedge \ldots \wedge b_{ik}) \to a_i)$$

知識は否定を含まない帰結の集まりとして表現されている．

こういう形の知識に対して反駁を行うには帰納法を使う．

```
let rec imply_horn (clauses : (string * string list) list)
    (goal : string list) =
  match goal with
    [] -> true
```

```
  | a :: ra ->
      List.exists
        (fun (a',bs) -> a = a' && imply_horn clauses (bs @ ra))
        clauses
```

ただし，依存関係に循環があると，上の関数が止まらない可能性がある．

練習問題 12.1 アトムの数が有限であることを利用し，計算が必ず止るように imply_horn を修正せよ．ヒント：goal と bs の関係を調べるといい．

述語論理

述語論理では，命題は対象を持つようになる．対象は項で表され，式は項を引数の取る述語から作られる．

項			
t	$::=$	x	変数
	$\mid$	c	定数
	$\mid$	$f(t,\ldots,t)$	関数
式			
A	$::=$	$p(t,\ldots,t)$	述語
P	$::=$	A	
	$\mid$	$\forall x.P$	全称
	$\mid$	$\exists x.P$	存在
	$\mid$	$\neg P \mid P \wedge P \mid P \vee P \mid P \rightarrow P$	

命題論理の法則に加えて，全称と存在が関係を持っている．

$$\neg\forall x.P = \exists x.\neg P$$

さらに，全称と存在は抽出できる．x は Q の中で自由に現われなければ，

$$\begin{aligned}(\forall x.P) \wedge Q &= \forall x.(P \wedge Q)\\ (\forall x.P) \vee Q &= \forall x.(P \vee Q)\\ (\exists x.P) \wedge Q &= \exists x.(P \wedge Q)\\ (\exists x.P) \vee Q &= \exists x.(P \vee Q)\end{aligned}$$

選言標準形および連言標準形は定義できるが，全称と存在が前に置かれる．

$$\forall x_1 \exists x_2 \ldots \forall x_n. \bigvee_i (\bigwedge_j A_{ij} \wedge \bigwedge_k \neg A'_{ik})$$

練習問題 12.2 命題論理と同様に，選言標準形に変換する関数を定義せよ

述語論理のホーン節

述語論理ではホーン節が少し一般化される．

$$\forall x_1 \ldots x_l.(A_1 \wedge \ldots \wedge A_k \rightarrow A_0)$$

命題論理と同様に反駁が定義されるが，そのときの変数の扱いを考えなければならない．OCaml 同様，マッチングで変数の割り当てを決めることもできるが，それだとあらかじめ探したい帰結を完全に決めておかないといけない．実際には，目標の中に変数を残した方が便利．例えば，適当な知識を与え，$\exists x.grandfather(Hugo, x)$ のような目標に対して，具体的な x の可能な値を知りたい．

単一化

二つの項を与えられると，変数に代入を行うことで同じものになるかどうかを判定し，できるなら同じものにする最も一般的な代入を返す.

```
type term =
  | V of string                                                    (* 変数 *)
  | C of string                                                    (* 定数 *)
  | F of term list                                                 (* 関数 *)

let rec subst s = function
    V x when List.mem_assoc x s -> List.assoc x s
  | F tl -> F (List.map (subst s) tl)
  | t -> t

let rec occurs x = function                          (* V x が t に出現するか *)
    V y -> x = y
  | C _ -> false
  | F tl -> List.exists (occurs x) tl

exception Cannot_unify of term * term

let rec unify s = function
    [] -> s
  | (t1, t2) :: others ->
      match (subst s t1, subst s t2) with
      | (V x, t) | (t, V x) when not (occurs x t) ->
          let s' = (x, t) :: List.map (fun (y,u) -> (y, subst [x,t] u)) s in
          unify s' others
      | (C s1, C s2) when s1 = s2 ->
          unify s others
      | (F tl1, F tl2) when List.length tl1 = List.length tl2 ->
          unify s (List.combine tl1 tl2 @ others)
      | (t1, t2) ->
          raise (Cannot_unify (t1, t2))
;;

# unify [] [F[V"a"; C"b"; V"c"], F[V"b"; V"c"; V"d"]];;
- : (string * term) list = [("d", C "b"); ("c", C "b"); ("a", V "b")]
# unify [] [F[V"a"; C"b"; V"c"], F[V"b"; V"c"; C"d"]];;
Exception: Cannot_unify (C "b", C "d").
```

練習問題 12.3 単一化を用いたホーン節の反駁を定義せよ．成功したときに元の目標に代入をかけた結果を返せ.

ホーン節の変数を使う度に新しいものにしなければならない．そのための補助関数 genvar を以下に定義する.

```
let genvar =
  let counter = ref 0 in
  fun s -> incr counter; V (s ^ "/" ^ string_of_int !counter)
```