归纳法推理中的子句简化策略*

李卫华 张 黔 承雪琦

(武汉大学计算机科学系 武汉 430072)

摘要 本文介绍归纳法推理系统中的简化策略.系统推理能力在很大程度上取决于系统简化待证公式的能力.本文从定义的类型规定出发,描述了如何计算并利用类型集信息来简化子句,以及如何在运用重写策略的基础上,完成对各种子句的简化.该系统已在微机上用编译LISP语言实现.

关键词 类型集,类型规定,归纳法推理.

在归纳法推理系统中[1~5],作用各种推理策略之后,都要借助子句简化策略简化推理过程中产生的中间结果.特别是,系统要把归纳法结论转换成与归纳法假设类似的表达式,从而可用归纳法假设来简化归纳法结论.

1 子句的简化

子句的简化建立在对文字重写的基础之上.设子句的形式为{new_1… new_n old_1 old_2… old_m},其中 new_1, …, new_n 为已重写文字,下面重写文字 old_1.这时假定除 old_1之外的任何文字均为假,若文字形为(not term),则假定 term 为真.然后根据这些假定重写 old_1,得到的结果为 result,若 old_1形为(not $oldterm_1$),则否定 result,并令 val 为最终结果.

若 val 为 T,则子句为真,不必再重写后面的文字;若 val 为 F,则从子句中删去 old_1,并继续重写 old_2;否则用 val 替代 old_1,并继续重写 old_2.

```
(define (simplify-clause1 tail new-clause lits-ignored-by-linear i)
  (define res ta) (define rr ())
  (define curlit current-lit) (define curatm current-atm)
  (set! ta ()) (set! current-lit ()) (set! current-atm ())
  (call/cc (lambda (return)
    (let ((segs ()) (neg-hyps ()) (branches ()))
      (cond ((null? tail) (set! ta res) (return (list new-cl)))
            (t (print-to-display 'simplify-clause i ())
               (set! current-lit (set! current-atm (car tail)))
```

* 本文研究得到国家863高科技项目和国家教委跨世纪优秀人才基金资助.作者李卫华,1952年生,教授,博士导师,主要研究领域为人工智能,知识工程,多媒体软件.张黔,女,1973年生,硕士生,主要研究领域为归纳法推理,多媒体软件.承雪琦,女,1973年生,硕士生,主要研究领域为归纳法推理,多媒体软件.

本文通讯联系人:李卫华,武汉430072,武汉大学计算机科学系

本文1995－08－30收到修改稿

```
(match current-atm (not current-atm));重写文字形为(not term)
(set! lits-ignored-by-linear
    (cons current-lit lits-ignored-by-linear))
(set! ta (ta-cl new-cl))
(when (eq? ta 'contradict) (set! ta res) (return ()))
(set! ta (ta-cl (cdr tail)))
(when (eq? ta 'contradict) (set! ta res) (return ()))
(init-linearize-assumpts-stack)
(push-linearize-assumpts-frame)
(set! fnstack ()) (set! *argl* ()) (set! *fnn* ())
(set! resstack ()) (set! resflg ()) (set! *con* ())
(set! val (rewrite current-atm () type-alist '? 'iff ())) ;重写文字
(when (not (eq? current-lit current-atm))
    (set! val (negate-lit val)))
(set! linear-assumpts (pop-linearize-assumpts-frame))
(set! neg-hyps (for hyp in linear-assumpts save (negate-lit hyp)))
(set! branches (clausify val)) ;重写结果的子句化
(set! segs (conjoin-cl-sets
             (for seg in branches save (disjoin-cls neg-hyps seg))
             (do ((hyp linear-assumpts (cdr hyp))
                  (cl (add-literal (pegate-lit current-lit) () ()))
                  (ans () ans))
                 ((null? hyp) (reverse ans))
                 (set! ans (cons (add-literal (car hyp) cl ()) ans)))))
(set! rr (do ((seg segs (cdr seg)) (lans () lans))
             ((null? seg) lans)
             (set! lans
                (nconc (simplify-clause1 (cdr tail) (append new-cl (car seg))
                               (if (match branches ())
                                   lits-ignored-by-linear
                                   (cdr lits-ignored-by-linear))
                               (add1 i));简化剩余子句
                       lans))))
(set! ta res) (set! current-lit curlit) (set! current-atm curatm)
(return rr)))))))
```

2　if 表达式的简化

若重写文字的结果 val 中含有 if 表达式,则要用以下2条规则去掉所有的 if 表达式:

将下面重写规则反复运用到 val 中,从而使所有的 if 移到其它函数符外面:

(equal (f x_1 … (if test left right) … x_n)

　　(if test (f x_1 … left … x_n) (f x_1 … right … x_n)))

因为(if test left right)等价于(and (implies test left) (implies (not test) right)),所以子句集{cl_1,… (if test left right) …,cl_n}等价于子句集{cl_1,… (not test) left …,cl_n}和子句集{cl_1,… test right …,cl_n}.运用这条规则,可将重写结果中的所有 if 表达式去掉.

```
(define (strip-ifs1 term topflg negate-flg)
  (define ans1 ()) (define ans2 ()) (define ans3 ())
  (define ans ()) (define lst ()) (define new-cl ())
  (cond ((variable? term)
```

```
  (list (cons (if negate-flg (negate-lit term) term) ())))  ;negate-lit 是求项的否定式
 ((qquote? term)
  (cond (topflg (cond ((equal? term false) (if negate-flg () (list (cons false ()))))
                      (negate-flg (list (cons false ())))
                      (t ())))
        (negate-flg (list (cons (if (equal? term false) true false) ())))
        (t (list (cons term ())))))
 ((eq? (ffn-symb term) 'if)  ;term 为 if 表达式
  (cond ((and topflg (or (and (not negate-flg) (equal? (fargn term 3) false))
                         (and negate-flg (equal? (fargn term 3) true))))
         (append (for pair in (strip-ifs1 (fargn term 1) topflg ()) when
                      (not (equal? (set! new-cl
                                         (add-literal (pegate-lit (car pair)) (cdr pair) t))
                                   t-clause))
                      save (cons false new-cl))
                 (strip-ifs1 (fargn term 2) topflg negate-flg)))
        ((and topflg (or (and (not negate-flg) (equal? (fargn term 2) false))
                         (and negate-flg (equal? (fargn term 2) true))))
         (append (for pair in (strip-branches1 (fargn term 1) topflg t) when
                      (not (equal? (set! new-cl
                                         (add-literal (pegate-lit (car pair)) (cdr pair) t))
                                   t-clause))
                      save (cons false new-cl))
                 (strip-ifs1 (fargn term 3) topflg negate-flg)))
        (t (set! ans1 (strip-ifs1 (fargn term 1) () ()))
           (set! ans2 (strip-ifs1 (fargn term 2) topflg negate-flg))
           (set! ans3 (strip-ifs1 (fargn term 3) topflg negate-flg))
           (for pair in ans1 do
             (begin
               (for pair2 in ans2 when
                 (not (equal?
                        (cdr (set! ans
                                   (cons (car pair2)
                                         (disjoin-cls (cdr pair)
                                           (add-literal (negate-lit (car pair)) (cdr pair2))))))
                        t-clause)) do (set! lst (cons ans lst)))
               (for pair3 in ans3 when
                 (not (equal?
                        (cdr (set! ans
                                   (cons (car pair3)
                                         (disjoin-cls (cdr pair)
                                           (add-literal (pegate-lit (car pair)) (cdr pair3))))))
                        t-clause))
                 do (set! lst (cons ans lst)))))
           lst)))
 (t (for pick in (all-picks (for arg in (fargs term) save (strip-ifs1 arg () ())))
         ;对表达式 term 的参量递归调用本函数以删除参量中的 if 表达式
         save (cons (if negate-flg
                        (dumb-negate-lit
                          (scons-term (ffn-symb term) (for pair in pick save (car pair))))
                        (scons-term (ffn-symb term) (for pair in pick save (car pair))))
                    (do ((pair pick (cdr pair)) (ans () ans))
                        ((null? pair) ans)
                        (when (equal? ans t-clause)
                              (set! ans (disjoin-clauses (cdr pair ) ans)))))))))
```

在删除 if 表达式时，会将一个子句分裂为2个. 若某归结式包括这2个子句，则保留该归

结式并丢弃被包括子句.例如,通过上面工作,系统可将子句:
{(if $test_1$(if $test_2$ $left_2$ $right_2$) (if $test_2$ $left_2$ $right_3$))}转换成以下4个子句:
①{(not $test_1$) (not $test_2$) $left_2$};②{(not $test_1$) $test_2$ $right_2$};③{$test_1$(not $test_2$) $left_2$};④{$test_1$ $test_2$ $right_3$}.其中①式表明,给定$test_1$时,要推出(or (not $test_2$) $left_2$);③式表明,给定(not $test_1$)时,要推出(or (not $test_2$) $left_2$).也就是说,对任意前提,均需推出(or (not $test_2$) $left_2$).于是原子句可转换为3个子句:①{(not $test_2$) left};②{(not $test_1$) $test_2$ $right_2$};③{$test_1$ $test_2$ $right_3$}.

下面的程序用来检查子句包括.

```
(define (almost-subsumes-loop lst) …) (略)
```

3　类型集及其计算

设系统已引入了数据类型{false},{true},{add1},{pack},{cons},{minus}等.设type-set-f(1),type-set-t(2),type-set-numbers(4),type-set-litatoms(8),type-set-cons(16),type-set-negatives(32)分别代表相应的数据类型的全部对象组成的集合,并简记为数2^n.因为类型集都是有穷集,系统实现时将对之作二进制处理.这样求2类型集的并集只要计算它们的逻辑或即可,求2类型集的交集只要计算它们的逻辑与即可.

定义1. 称集合 type-set-f(2),type-set-f(1),type-set-numbers(4),type-set-litatoms(8),type-set-cons(16),type-set-negatives(32),…,均为类型.

定义2. 由多个类型组成的集合称为一个类型集.设由全部类型组成的集合记为type-set-unknown(−1).若typeset为一个类型集,且在项term中变量的任一解释下,term的值属于typeset的一个类型,则称项term具有类型集typeset.

定义3. 称偶对〈ts,args〉为函数f的一个类型规定是指:

ts是一个类型集;args是f之形参集的一个子集;当项t_1,…,t_n分别有类型集s_1,…,s_n时,则(f t_1 … t_n)的类型集为:使得f中第i个形参在args中的那些s_i的并集与ts的并集.

定义4. 由一类型集和变量的有穷集所组成的偶对称为定义类型集.

定义5. 称项具有定义类型集〈ts,s〉是指:

ts是一个类型集;s是一个变量集;s中的每一变量在这个项中出现,在该项中变量的任一解释下,此项之值或为ts的一个元素或为在同一解释下s的值.

引入新函数f时,系统采用下述方法确定其类型规定:

假定f有类型规定〈{},{}〉,计算f之定义体的定义类型集,再对先前假定的类型规定与最近计算的定义类型集的偶对求并集作为f的类型规定.重复上述操作,直到所计算的定义类型集为假定类型规定的偶对的子集,并将此类型规定作为f的最终类型规定.

例如,系统引入plus和append的定义时,生成的类型规定分别为(type-prescript-l ((plus 4 () ()))和(type-precript-l ((append 16 () T))).它表明函数(plus x y)的类型规定为〈4{type-set-numbers},{}〉;(append x y)的类型规定为〈16{type-set-cons},{y}〉.

```
(define (put-type-prescript fn)
  (define old-type-prescript ()) (define new-type-prescript ())
  (define body ()) (define formals ()) (define type-l ()) (define ans ())
  (define type-prescript-name-&-pair ())
```

```
(call/cc (lambda (return)
  (set! type-l ta)
  (set! body (getprop fn 'sdefn)) ;取 fn 的定义特性
  (set! formals (cadr body)) ;取定义的形参
  (set! body (caddr body)) ;取定义的定义体部分
  (set! ta (for arg in formals save (cons arg (cons 0 (list arg)))))
  (set! old-type-prescript (cons 0 (for arg in formals save ())))
  (set! type-prescript-name-&-pair (cons fn old-type-prescript))
  (add-fact fn 'type-prescript-l type-prescript-name-&-pair)
  (let ((go 'loop) (p 1))
    (call/cc (lambda (loop) (set! go loop))) ;设置跳转标志
    (set-cdr! type-prescript-name-&-pair old-type-prescript)
    (set! ans (defn-type-set body))
    (set! new-type-prescript
      (cons (car ans) (for arg in formals save (if (memq arg (cdr ans)) t ()))))
    (cond ((equal? old-type-prescript new-type-prescript)
           (set! ta type-l) (return ()))
          ((and (logsubset? (car new-type-prescript) (car old-type-prescript))
                (do ((flg1 (cdr new-type-prescript) (cdr flg1))
                     (flg2 (cdr old-type-prescript) (cdr flg2)))
                    ((or (null? flg1) (not (or (not (car flg1)) (car flg2))))
                     (if (null? flg1) t ()))))
           (error "新类型规定是旧类型规定的真子集" ())
           (set! ta type-l) (return ())))
    (set! old-type-prescript ;重新修改类型规定
      (cons (%logior (car old-type-prescript) (car new-type-prescript))
            (do ((flg1 (cdr old-type-prescript) (cdr flg1))
                 (flg2 (cdr new-type-prescript) (cdr flg2)) (lans () lans))
                ((null? flg1) (reverse lans))
                (set! lans (cons (or (car flg1) (car flg2)) lans)))))
    (go 'loop))))
```

类型集的计算采用下述方法获得：

·变量的类型集为 type-set-unknown.

·新引入定义函数的类型集

对于新定义的函数，系统先将其类型规定放入 type-prescript-l 特性中.

若类型规定中的参量部分非空，则将这些参量的类型集与类型规定中的类型集的并集作为此函数的类型集；否则直接返回类型规定中的类型集.

·识别符表达式的类型集

对识别符表达式(r term)，先计算参量 term 的类型集 ts.

若 ts 为{r}，则表达式为真；若 ts 中不含{r}，则表达式为假；否则表达式的结果为布尔类型(T 或 F). 假定表达式为真时，将 term 具有类型集{r}加入假定中；假定表达式为假时，将 term 具有类型集 ts−{r}加入假定中.

·equal 表达式的类型集

若表达式形为(equal left right)，则先计算参量 left，right 的类型集 ls，rs.

若 ls 和 rs 的交集为空，则表达式为假；若 ls 与 rs 相等，且 ls 为孤元类型集，则表达式为真；否则表达式的结果为布尔类型(可为 T 或 F).

假定表达式为真时，将(equal left right)具有类型集{type-set-t}加入假定中，并将(equal right left)具有类型集{type-set-t}也加入假定中，同时认为 left，right 的类型集均为

ls 和 rs 的交集;

假定表达式为假时,将(equal left right)具有类型集{type-set-f}加入假定中,并将(equal right left)具有类型集{type-set-f}也加入假定中,同时若 left 为一孤元,则 right 具有类型集 rs－ls,反之亦然.

·not 表达式的类型集

若表达式形为(not term),则先计算参量 term 的类型集 ts.

若 ts 与{type-set-f}相等,则表达式为真;若 ts 中不含{type-set-f},则表达式为假;否则表达式的结果为布尔类型(T 或 F).

·if 表达式的类型集

若表达式形为(if test left right),则先判断 test 的取值情况.

若 test 必为真,则 left 的类型集即为表达式的类型集;若 test 必为假,则 right 的类型集即为表达式的类型集;否则表达式的类型集为 left 与 right 类型集的并集.

```
(define (type-set term)
  (define pair ()) (define type-arg1 ()) (define type-arg2 ()) (define arg1 ()) (define arg2 ())
  (when (not type2flg) (set! ttype ta))
  (cond ((set! tmp (assoc term ttype)) (cdr tmp))
        ((variable? term) type-set-unknown) ;term 为变量
        ((qquote? term) (car (type-prescript (fn-symb0 (cadr term))))) ;term 形为(quote …)
        ((set! pair (assq (ffn-symb term) ra)) ;term 的函数符为外壳识别符
         (set! type-arg1 (type-set (fargn term 1)))
         (cond ((zero? (%logand type-arg1 (cdr pair))) type-set-f)
               ((logsubset? type-arg1 (cdr pair)) type-set-t)
               (t type-set-b)))
        ((match term (equal? arg1 arg2)) ;term 为 equal 表达式
         (set! type-arg1 (type-set arg1))
         (set! type-arg2 (type-set arg2)) ;计算参量的类型集
         (cond ((zero? (%logand type-arg1 type-arg2)) type-set-f)
               ((and (equal? type-arg1 type-arg2) (member type-arg1 singleton-type-sets))
                type-set-t)
               (t type-set-b)))
        ((match term (not arg1)) ;term 为 not 表达式
         (set! type-arg1 (type-set arg1)) ;计算参量的类型集
         (cond ((equal? type-arg1 type-set-f) type-set-t)
               ((not (logsubset? type-set-f type-arg1)) type-set-f)
               (t type-set-b)))
        ((eq? (ffn-symb term) 'if) ;term 为 if 表达式
         (assume-t-f (fargn term 1)) ;对 if 表达式的测试判断真假
         (cond (must-be-t (type-set (fargn term 2))) ;测试为真,计算左部的类型集
               (must-be-f (type-set (fargn term 3))) ;测试为假,计算右部的类型集
               (t (%logior (type-set2 (fargn term 2) t-ta)
                           (type-set2 (fargn term 3) f-ta)))))
        ((set! tmp (type-prescript (ffn-symb term))) ;取 term 函数符的类型规定
         (%logior (car tmp)
                  (do ((arg (fargs term) (cdr arg)) (flg (cdr tmp) (cdr flg)) (lans 0 lans))
                      ((null? arg) lans)
                      (when (car flg) (set! lans (%logior lans (type-set (car arg))))))))
        (t type-set-unknown)))
```

参考文献

Boyer R S, Moore J S 著. 李卫华译. 计算逻辑. 计算机工程与应用,1982,154(4),155(5),1983,169(7).

2 李卫华,张黔,刘娟,石自力.归纳法推理系统.计算机学报,1996,19(3):230～236.
3 李卫华,张黔,张亮,刘娟.归纳法模式的自动生成.软件学报,1996,7(3):168～174.
4 李卫华,张黔,韩波.归纳法推理中的项重写策略.软件学报,1996,7(增刊):565～571.
5 李卫华,张黔,龙泉.归纳法推理中的各种推理策略.软件学报,1996,7(增刊):551～557.
6 李卫华,陈兆乾,潘金贵.人工智能程序设计.北京:科学出版社,1989.

CLAUSE SIMPLIFY STRATEGY IN INDUCTION INFERENCE

Li Weihua Zhang Qian Cheng Xueqi

(*Department of Computer Science Wuhan University Wuhan* 430072)

Abstract This paper discusses the clause simplify strategy of induction inference system. The inference ability of the system mainly depends on the ability of simplifying the clauses. Starting from the type prescript of the definition, this paper introduces how to compute and use the type set information to simplify the clauses, how to use the rewrite strategy to simplify the clauses. The system has been implemented by using compiler LISP on micro computer.

Key words Type set, type prescript, induction inference.