# 归纳法推理中的项重写策略*

李卫华 张黔 韩波

(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项重写策略.该策略根据不同的待重写项term,分别运用公理、重写引理、函数定义、项重写规则等重写项term,以期得到一个更接近推理目标的已重写项.这一策略已在微机上用编译LISP语言实现.

**关键词** 项重写,归纳法推理.

归纳法推理系统由2大部分组成.一部分是对知识库的维护,包括引入新数据类型,增加已证引理,检查新增公理和引理与知识库的一致性,引入满足定义条件的新函数等.另一部分是对待证公式施以若干推理策略,如项重写、子句简化、分元符删除、交融、推广、无关式删除、归纳法模式的自动生成等.[1~6]本文介绍项重写策略.

## 1 项的重写

给定变量—值结合表va,项—类型集结合表ta,函数rewrite完成对项term的重写.包括对变量、显式值(quote…)、if表达式(if test left right)、equal表达式(equal left right)、识别符表达式(r x)、函数调用表达式(fn $arg_1$ … $arg_n$)等项term的重写.

```
(define (rewrite term va ta objective id-iff defn-flg)
  (cond ((variable? term) ;重写变量
         (rewrite-solidify (if (set! tmp (assq term va)) (cdr tmp) term)))
        ((qquote? term) term) ;重写显式值(quote …)
        ((eq? (ffn-symb term) 'if) ;重写(if …)
         (rewrite-if (rewrite (fargn term 1) va ta '? 'iff ()) (fargn term 2) (fargn term 3) ta))
        ((set! tmp (not-to-be-rewritten? term va)) (rewrite-solidify tmp))
        (t (let ((args ()) (fn ()))          ;重写函数调用中的实参
             (set! args (for arg in (fargs term) save (rewrite arg va ta '? 'id ())))
             (if (and (for arg in args always (quote? arg))
                      (if (getprop (getprop (ffn-symb term) 'lisp-code) 'expr)
                          (set! fn (getprop (getprop (ffn-symb term) 'lisp-code) 'expr))
                          (set! fn (getprop (ffn-symb term) 'lisp-code)))) ;取函数符的机内表示
                 (begin (push-lemma (ffn-symb term))   ;将此函数作用于其对应实参上
                        (list 'quote (apply (eval fn) (for arg in args save (cadr arg)))))
```

---

* 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.韩波,1972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—31收到修改稿

```
    (rewrite-with-lemmas      ;用引理重写
      ;重写(equal …)表达式或识别符表达式(r …)等
      (rewrite-type-pred (fcons-term (ffn-symb term) args))))))))
```

## 2 变量的重写

若变量—值结合表 va 中存在着元素:(term . valt),则重写项 term 的结果为重写 valt 的值;若项—类型集结合表 ta 中存在着元素:((equal term right) . true),则重写项 term 的结果为 right;若 ta 中存在某元素为(term . typet),则将重写 typet 的结果作为重写 term 的结果;否则直接返回 term 本身.

```
(define (rewrite-solidify term)
  (define lit ()) (define temp ()) (define lhs ()) (define rhs ())
  (cond ((quote? term) term) ;(quote …)表达式的重写结果仍为自身
        ((and (not (variable? term)) (eq? (ffn-symb term) 'if))
         term)          ;变量在 va 中对应的项为 if 表达式
        ((do ((pair ta (cdr pair)))
             ((or (null? pair) (and (eq? (cdar pair) type-set-t) ;若 ta 中含((equal term rhs) . true)
                                    (set! tmp (caar pair))       ;则把 term 重写为 rhs.
                                    (match tmp (equal lhs rhs))
                                    (equal lhs term)))
              (if (null? pair) () t)))
         rhs)
        ((and (set! tmp (assoc term ta)) ;由 objective 的取值判断类型集真假的意义
              (obj-table (cdr tmp) objective id-iff)))
        ((set! lit (for lit in lits-that-assumed-f when      ;项 term 为真或假
                        (cond ((equal lit term) (set! temp false))
                              ((complementary? lit term) (set! temp true))
                              (t ())))
                        return lit))
         (if (or (eq? id-iff 'iff) (eq? temp false) (boolean term))
             (begin (push-lemma lit) temp)
             term))
        (t term)))
```

## 3 if 表达式的重写

设表达式 term 形为:(if test left right),若 test 的重写结果 test′为 T,则返回对 left 的重写结果;若 test′为 F,则返回对 right 的重写结果;否则在 test′为真的假定下重写 left 得 left′,在 test′为假的假定下重写 right 得 right′,然后设法将下述规则作用到(if test′ left′ right′)上:(equal (if x y y) y),(equal (if x x f) x),(equal (if x t f) x)仅用于 x 为布尔量的情况.

```
(define (rewrite-if test left right ta)
  (define (rewrite-if1 test left right)
    (cond ((ident left right) left)      ;(if x y y)的重写结果为 y
          ((and (ident test left) (f-nonf? right) definitely-f) test);(if x x f)为 x
          ((and (ident true left) (f-nonf? right) definitely-f (boolean test)) test);(if x t f)为 x
          (t (fcons-term * 'if test left right))))
  (define res ()) (define rr ())
  ;(if (if test' 假 非假) left right)的重写结果为(if test' right left)
  (when (and (not (variable? test)) ;test 不为变量
             (not (qquote? test)) ;test 不为(quote …)表达式
```

```
            (eq? (ffn-symb test) 'if) ;test 形为(if test' left'right')
            (equal (fargn test 2) false) ;test 表达式的 left'为假?
            (f-nonf? (fargn test 3))
            (not definitely-f)) ;test 表达式的 right'不为假?
      (swap left right) ;交换 left 与 right
      (set! test (fargn test 1))) ;将 test 置成 test'
  (assume-t-f test) ;判断 test 的真假
  (cond (must-be-t ;(if 非假 left right)的重写结果为重写 left 的结果
          (good left (rewrite left va ta objective id-iff defn-flg)))
        (must-be-f ;(if 假 left right)的重写结果为重写 right 的结果
          (good right (rewrite right va ta objective id-iff defn-flg)))
        (t (set! res f-ta) (set! f-ta ()) ;在 test 为真的假定下重写 left
          (set! rr (good left (rewrite left va t-ta objective id-iff defn-flg)))
          (set! f-ta res)
          ;left,right 分别重写之后再运用相关规则
          (rewrite-if1 test rr ;下式在 test 为假的假定下重写 right
            (good right (rewrite right va f-ta objective id-iff defn-flg))))))
```

## 4　equal 与识别符表达式的重写

**定义1.** 2个项在定义上不相等是指,这2个项满足下述条件之一:它们的类型集不相交;它们是不同的显式值;一个项是底对象,另一个项是一外壳构元符的调用;一个项是一外壳构元符的调用,而另一个项在该调用中作为组分出现.

设表达式为:(equal left right),若 left 与 right 的重写结果恒等,则返回 T;若 left 与 right 在定义上不相等,则返回 F;否则设法将下述规则作用到该 equal 表达式上:

(equal (equal x t) x)　　仅用于 x 为布尔量的情况

(equal (equal x (equal y z)) (if (equal y z) (equal x t) (equal x f)))

(equal (equal f x) (if x f t)) (equal (equal x f) (if x f t))

(equal (equal (equal y z) x) (if (equal y z) (equal x t) (equal x f)))

若无规则可用,则返回用重写引理重写该表达式的结果.

设表达式为:(r term),其中 r 为一外壳识别符,若 term 的类型集为{r},则返回 T;若重写 term 的类型集不含 r,则返回 F;否则返回用重写引理重写该表达式的结果.

```
(define (rewrite-type-pred term)
  (define left ()) (define right ()) (define pair ()) (define typeset ())
  (cond ((or (variable? term) (qquote? term)) term)
        ((match term (equal left right)) ;term 形为(equal left right)
          (cond ((ident left right) true) ;left 与 right 恒等时返回 T
                ((not-ident left right) false) ;left 与 right 在定义上不相等时返回 F
                ((and (boolean left) (ident true right)) left) ;用规则(equal (equal x t) x)
                ;用规则(equal (equal x (equal y z)) (if (equal y z) (equal x t) (equal x f)))
                ((match right (equal & &))
                  (fcons-term * 'if right
                      (fcons-term * 'equal? left true) (fcons-term * 'if left false true)))
                ;用规则(equal (equal f x) (if x f t))
                ((ident left false) (fcons-term * 'if right false true))
                ;用规则(equal (equal x f) (if x f t))
                ((ident right false) (fcons-term * 'if left false true))
                ;用规则(equal (equal (equal y z) x) (if (equal y z) (equal x t) (equal x f)))
                ((match left (equal & &))
                  (fcons-term * 'if left
```

```
        (fcons-term * 'equal right true) (fcons-term * 'if right false true)))
    ((and (set! typeset (type-set left)) ;left,right 的类型集比较
          (do ((x ra (cdr x)))
              ((or (null? x) (equal type-set (cdar x)))
               (if (null? x) () t)))
          (equal typeset (type-set right))
          (not (btm-object-of-type-set typeset)))
     (let ((logic ()))
       (do ((dest (cdr (assq (car (for x in sa when
                                    (equal typeset (logbit (cdr x))) return x))
                             shell-pockets)) (cdr dest)))
           ((and logic (or (null? dest) (equal tmp false) (not (equal tmp true))))
            (cond ((null? dest) true)
                  ((equal tmp false) false)
                  ((not (equal tmp true)) term)))
           (set! tmp
             (rewrite (fcons-term * 'equal
                        (fcons-term * (car dest) 'left) (fcons-term * (car dest) 'right))
                      (list (cons 'left left) (cons 'right right))
                      ta '? 'id ()))))) ;递归地重写
      (t term)))
 ((set! pair (assq (ffn-symb term) ra)) ;识别符表达式(f …)的重写
  (set! typeset (type-set (fargn term 1)))
  (cond ((logsubset? typeset (cdr pair)) true)
        ((zero? (%logand typeset (cdr pair))) false)
        (t term)))
 (t term)))
```

## 5 函数调用表达式的重写

在归纳法推理过程中,系统经常遇到对已定义函数 f 的调用表达式,用函数 f 的定义体替代 f 的调用表达式常常有利于逼近推理目标.由于非递归函数的调用不会引起无限循环,故在推理时可直接用经形实替换后的函数定义体替代该函数调用.对于递归函数 f,设重写其定义体的结果为 val,若 val 和 f 满足以下任一条件,则可用 val 替代 f:

在 val 里 f 的调用中,每一参量已出现在待证推测中;在 val 里 f 的调用中,显式值参量比 f 调用表达式中的显式值参量多;在 val 里 f 的调用中,某被测度子集的符号复杂性比原始调用中 f 的被测度子集的符号复杂性小.

```
(define (rewrite-fncall *fn* *args*)
  (define rr ()) (set! reslemma lemma-stack)
  (set! lemma-stack lemma-stack)     (set! *typel* *ta*)
  (let ((value ()) (sdefn ()) (linearize-assumpts-stack linearize-assumpts-stack))
    (set! sdefn (getprop *fn* 'sdefn)) ;取 *fnname* 的定义特性
    (cond ((null? sdefn) (set! resflg (cons 'ddd resflg)) ;定义特性为空
           (set! rr (rewrite-solidify (cons-term *fn* *args*)))
           (set! *ta* *typel*) (set! lemma-stack reslemma)
           (set! resflg (cdr resflg)) rr)
          ((or (memq *fn* fnstack) (disabled? *fn*)) ;函数已在堆栈中
           (set! resflg (cons 'ddd resflg))
           (set! rr (rewrite-solidify (cons-term *fn* *args*)))
           (set! *ta* *typel*) (set! lemma-stack reslemma)
           (set! resflg (cdr resflg)) rr)
          (t (call/cc (lambda (rewrite-fncall)
```

```
(case (car resflg) ;定义不同的全局出口
      ("exit1" (set! exit-2 rewrite-fncall) (set! resflg (cons "exit2" resflg)))
      ("exit2" (set! exit-3 rewrite-fncall) (set! resflg (cons "exit3" resflg)))
      (else (set! exit-1 rewrite-fncall) (set! resflg (cons "exit1" resflg))))
(set! *argl* (cons *args* *argl*)) (set! *fnn* (cons *fn* *fnn*))
(set! *controller-complexities* ;设置控制符的复杂度
  (for mask in (getprop *fn* 'controller-pockets) save
    (do ((arg *args* (cdr arg)) (lans 0 lans))
        ((null? arg) lans)
        (when (begin0 (not (eq? (%logand mask 1) 0)) (set! mask (lsh mask -1)))
              (or (quote? arg) (set! value ()))
              (set! lans (+ (max-form-count (car arg)) lans))))))
(set! *con* (cons *controller-complexities* *con*))
(set! resstack fnstack) (set! fnstack (cons *fn* fnstack)) ;保存 fnstack 值
(push-lemma-frame) (push-linearize-assumpts-frame) ;将有关信息放入堆栈
(set! value
  (rewrite (caddr sdefn) ;递归地重写定义
      (do ((var (cadr sdefn) (cdr var)) (val *args* (cdr val)) (lans () lans))
          ((null? var) (reverse lans))
          (set! lans (cons (cons (car var) (car val)) lans)))
          ta objective id-iff t))
(cond
 ((null? (getprop *fn* 'induct-machine)) ;判断函数是否为递归函数
  (cond ((and (do ((x (cdr fnstack) (cdr x)))
                  ((or (null? x) (getprop (car x) 'induct-machine))
                   (if (null? x) t ())))
              (too-many-ifs *args* value)) ;项中 IF 表达式是否过多
         (pop-lemma-frame) (pop-linearize-assumpts-frame)
         (set! rr (rewrite-solidify (fcons-term (car *fnn*) (car *argl*))))
         (set! *fnn* (cdr *fnn*)) (set! *argl* (cdr *argl*))
         (when (not (equal resstack fnstack))
             (set! fnstack resstack) (set!resstack (cdr resstack)))
         (set! *ta* *typel*) (set! lemma-stack reslemma)
         (set! *controller-complexities* (car *con*)) (set! *con* (cdr *con*))
         (set! resflg (cdr resflg)) rr)
        (t (for x in (pop-linearize-assumpts-frame) do (push-linearize-assumpt x))
           (for x in (pop-lemma-frame) do (push-lemma x))
           (push-lemma *fn*) (set! rr value)
           (when (not (equal resstack fnstack));比较新老堆栈值
                 (set! fnstack resstack) (set! resstack (cdr resstack)))
           (set! *ta* *typel*) (set! lemma-stack reslemma)
           (set! *controller-complexities* (car *con*)) (set! *con* (cdr *con*))
           (set! resflg (cdr resflg)) (set! *fnn* (cdr *fnn*))
           (set! *argl* (cdr *argl*)) rr)))
 ((rewrite-fncall? *fn* value) ;判断可否用 value 替代 *fn*?
  (for x in (pop-linearize-assumpts-frame) do (push-linearize-assumpt x))
  (for x in (pop-lemma-frame) do (push-lemma x))
  (push-lemma *fn*) (set! rr value)
  (when (not (equal resstack fnstack))
      (set! fnstack resstack) (set! resstack (cdr resstack)))
  (set! *ta* *typel*) (set! lemma-stack reslemma)
  (set! *controller-complexities* (car *con*)) (set! *con* (cdr *con*))
  (set! resflg (cdr resflg)) (set! *fnn* (cdr *fnn*))
  (set! *argl* (cdr *argl*)) rr)
 (t (pop-lemma-frame) (pop-linearize-assumpts-frame)
    (set! rr (rewrite-solidify (cons-term *fn* *args*))) ;重写(*fn* *srgs*)
```

```
(when (not (equal resstack fnstack))
  (set! fnstack resstack) (set! resstack (cdr resstack)))
(set! *ta* *typel*) (set! lemma-stack reslemma) ;恢复变量原值
(set! *controller-complexities* (car *con*))
(set! *con* (cdr *con*))
(set! resflg (cdr resflg)) (set! *fnn* (cdr *fnn*))
(set! *argl* (cdr *argl*)) rr)))))))
```

## 6 用引理重写项

重写引理由2种方法生成,通过定义新数据类型时引入;引用已证定理.[2]

设项(f $t_1$ … $t_n$)中参量 $t_i$ 业已重写,查找一条已知引理,其结论形为(equal left right),且(f $t_1$ … $t_n$)是 left 在某种代换 s 下的一个例示. 若找到这条引理,就用 s 例示该引理中的假设,并递归地重写每一假设,若每一假设均重写为非假,就用被例示的 right 替代(f $t_1$ … $t_n$),最后递归地重写例示后的 right.

在运用上述方法重写待证公式时,有时可能会出现"死循环". 为此,引入下述定义:

定义2. 若一重写规则左部是右部的例示,且右部也是左部的例示,则称此规则为可置换规则. 对可置换规则,系统只允许将项转换为重写顺序较小的项.

系统用 term-order 来判断项1与项2中谁的重写顺序大:

```
(define (term-order t1 t2)
  (define form-count1 (form-count t1)) ;计算 t1 中的函数个数
  (define No-of-Vars1 No-of-Vars) ;计算 t1 中的变量个数
  (define form-count2 (form-count t2)) ;计算 t2 中的函数个数
  (define No-of-Vars2 No-of-Vars) ;计算 t2 中的变量个数
  (cond ((<? No-of-Vars1 No-of-Vars2) t) ;变量个数少的项重写顺序小
        ((<? No-of-Vars2 No-of-Vars1) ())
        ((<? form-count1 form-count2) t) ;函数个数少的项重写顺序小
        ((<? form-count2 form-count1) ())
        (t (lexorder t1 t2)))) ;字母顺序小的项重写顺序小
```

定义3. 递归地用引理去建立其它引理之假设的方法称为反向链.

定义4. 称 T,F,底对象,把外壳构元符作用到满足类型限制的显式值后所得的结果为显式值.

定义5. 若 new 与 old 满足以下条件之一,则称 new 比 old 复杂:

·old 是一个变量且严格出现在 new 中;

·old 和 new 都不是变量,且满足以下任一条件:

·new 和 old 有不同的函数符,且 new 的某一子项比 old 复杂或恒等于 old;

·new 的某参量比 old 的相应参量复杂,且 new 的任一参量均不为变量或显式值,除非 old 的相应参量均为变量或显式值,此外 old 的任一参量都不比 new 的相应参量复杂.

定义6. 若 new 与 old 满足以下条件之一,则称 new 是 old 的一个精致:

·new 恒等于 old;

·new 中函数符的出现次数大于或等于 old 中函数符的出现次数,且 new 比 old 复杂.

在推理一个待证公式时可能会出现无限反向链的情况. 为避免无限反向链的出现,系统采取如下方法:用表 L 记录当前试图证明的那些假设的否定式. 若想用重写规则 r,且 r 的

一个假设为new,若L中含new,则假定new为真,且不再证明;若L含new的否定式,则得知它正在循环,于是放弃运用r;若new的原子为某假设原子的精致,也得知它正在循环;以上条件均不满足时,将new的否定式存入L中,并递归地重写new.

上述启发式方法构成了用引理重写项的工作:

(define (rewrite-with-lemmas term) …) (略)

## 参考文献


1 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,刘娟等.归纳法推理系统.计算机学报,1996,19(3):230～236.

3 李卫华,张黔,张亮等.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,承雪琦.归纳法推理中的子句简化策略.软件学报,1996,7(增刊):558～564.

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.


# TERM REWRITING STRATEGY IN INDUCTION INFERENCE


Li Weihua　Zhang Qian　Han Bo

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



**Abstract**　This paper discusses the term rewriting strategy in induction inference system. Axioms, rewriting lemmas, function definitions, term rewriting rules are used to rewrite the term to be rewritten. The system will get a rewritten term which approaches to the inference goal. This strategy has been implemented by using compiler LISP on micro computer.



**Key words**　Term rewriting, induction inference.