ICOT Technical Memorandum: TM-0101

TM-0101

Some Experiments on EKL

Masami Hagiya and Susumu Hayashi
(Kyoto University)

March, 1985

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

Some Experiments on EKL

Masami Hagiya
Susumu Hayashi

Research Institute for Mathematical Sciences
Kyoto University

ABSTRACT

Some experiments on EKL are presented. EKL is an interactive proof checker developed at Stanford University to formalize mathematical proofs. The Symbolics version of EKL was ported to Common Lisp by the authors, and under the ported version, some proofs on linear algebra, quick sort algorithm, synthetic differential geometry, non-standard analysis have been made.

1. Introduction

EKL is an interactive proof checker developed by Jussi Ketonen and Joseph S. Weening of Stanford University with the intent of formalizing proofs in mathematics [Ketonen & Weening 84, 84a]. The most characteristic feature of EKL is its higher order language and rewriter. Using the higher order language, one can easily express various notions in mathematics. The rewriter of EKL is a very powerful and natural proof procedure; most of the lines in the EKL proofs are usually performed by the rewriter. Originally, EKL was developed in MacLisp on TOPS-20 and later ported to ZetaLisp on Symbolics 3600 by the developers themselves.

We got the source of both versions of EKL from Ketonen through Prof. Sato of the University of Tokyo, who visited Stanford from July to December 1984, and ported the Symbolics version of EKL to Kyoto Common Lisp (a Common Lisp implementation developed at our institute) running on ECLIPSE/MV of out institute. On porting the Symbolics version to Kyoto Common Lisp, the symbols represented by the special characters of Symbolics were replaced by those of the MacLisp version (≠ was replaced by !=).

The original source of EKL amounts to about ten thousand lines of code; after ported to Kyoto Common Lisp and compiled, the system occupies about 3.8 mega bytes of memory (including the LISP system).

The first experiment is the proof of the associativity of matrix product. It is done as a preliminary experiment for the CAP (Computer Aided Proof) Project [Furukawa & Yokoi 84] of the ICOT Working Group 5, in which a proof checker of linear algebra at undergraduate level is being constructed. The second experiment is the verification of the quick sort algorithm on lists. The first one mainly concerns with the manipulation of equations, while the second one does many case analyses and applies induction schemata

from time to time; the first one is a proof in mathematics, while the second one is a correctness proof of a program. We think that these two experiments completely different in nature expose the features of EKL from many respects.

In the last section, some small experiments related to analysis (synthetic differential geometry and non-standard analysis) are presented, in which some of the readers will have much interest.

2. Comments and Bugs

EKL would be one of the best existing proof checkers as far as we know. It has, however, no advanced devices like heuristics of Boyer-Moore theorem prover or tactics of Edinburgh LCF. EKL has a decision procedure DERIVE for a subtheory of predicate logic, but it was not so useful to generate proof lines in our experiments. (Indeed, in the experiment in Chapter 3, it was used only as the rule of introduction of universal quantifier.) Nontheless, EKL does very well its task, proof generations in mathematical theories, by the aid of its higher order language and rewriter. But we found some shorthands and bugs in the current versions of EKL, which we will present below.

A) When the term PREDICATE in an option (SORT PREDICATE) of a rewriter is a lambda-term, then the rewriter does not work.

B) When DERIVE fails EKL does not return enough messages about it. So the user cannot realize why it failed. For example, DERIVE tells you nothing except its failure, when you try to introduce a universal quantifier and there are some free variables in assumptions.

C) If the user carelessly declare types of terms or associativity of operators, then the theory might be contradictory. See the following example:

```
1. (define dream !all x.dream(x) iff not x = x! nil)

2. (decl unicorn (syntype constant) (sort !dream!))

3. (derive !dream(unicorn)! () nil)

4. (rw 3 (open dream))
;false
```

This kind of contradiction is inevitable, but EKL should be more careful about declarations.

D) Type polymorphism is practically not available. EKL has variable type. But it does not work properly.

E) EKL does not have the feature of bounded bindop like summation and bounded quantifies. By bounded bindop, we mean the following constructor:

```
BOUNDED_BINDOP IDENTIFIER ... IDENTIFIER . TERM. TERM
```

where the first TERM is not in the scope of the bound variables. Lacking this feature, we were forced to use lambda-notation in the definition of summation in Chapter 3. If EKL had had this feature, the proofs about summation would have been much more natural than the current proofs.

F) Most EKL commands deriving lines do a lot of things at a time. This is the reason why proofs in EKL are short, and it is one of the most remarkable features of EKL. Nontheless, we sometimes wished to do only one step of inference at a time, since EKL did too much more than we wished. For example, CI command does unnecessary rewriting. Assume you have proved

```
A = B |- C[A] = D.
```

If you discharge A = B by CI, then resulting line consists

```
A = B imp C[B] = D
```

instead of the desired line

```
A = B imp C[A] = D.
```

whenever B is simpler than A.

G) Introduction of universal quantifier of higher type fails. See the following example:

```
1. (assume |all f i.p(f(i))|)
|deps: (1)

2. (derive |p(f(i))| *)
|deps: (1)

3. (derive |all f i.p(f(i))| *)
| failed to derive all f i.p(f(i))
```

This bug appeared in both of the Symbolics 3600 version and the TOPS-20 version.

H) DERIVE failed to derive an arbitrary formula from a line consisting false. See the following sessions:

```
1. (assume |false|)
|deps: (1)
```

```
2. (derive |p| *)
; failed to derive p

2. (derive |false imp p|)

3. (derive |p| (1 *))
:deps: (1)
```

1) The rewriter of the Symbolics 3600 version has the following bug:

```
1. (decl times (type |(ground,ground,(ground*)):ground|)
   (syntype constant) (infixname *) (associativity both)
   (bindingpower 935))

2. (axiom |all n m.n*m = m*n|)

3. (trw
    |lambda j0 d1 d2.(lambda i k.sum(j0,lambda j.d1(i,j)*d2(j,k)))|
    (use 2))
:(lambda j0 d1 d2.(lambda i k.sum(j0,lambda j.d1(i,j)*d2(j,k)))) =
:(lambda j0 d1 d2.(lambda i k.sum(j0,lambda j.d2(j,k)*d1(i,j))))
```

Note that the last rewriting should not do anything, since d1 is simpler than d2. This bug does not appear in the TOPS-20 version.

3. Matrix

3.1. Data structure

This chapter presents the proof of the associativity of matrix product. We carried out the proof from the scratch. In fact, we used only properties of natural numbers and scalars as axioms. As noted in Chapter 2, we could not use bounded bindop to formulate summation, so it is represented as a function with two arguments sum(n,f), where n is a natural number and f is a function from natural number to scalar as follows:

```
(decl series (syntype constant) (type |(ground:ground):truthval|))

(define series |series=lambda ff. all i,scalar(ff(i))|)

(decl sum (syntype constant) (type |(ground,(ground:ground)):ground|))

(decl (f g h) (type |ground:ground|) (sort |series|))
```

```
(defax sum
 !all ff.(sum(0,ff)=szero) &
  all i ff.(sum(i',ff)=(ff(i') ++ sum(i,ff)))!).
```

A matrix is represents as a triple <column,row,mtbody>, where column and row are natural numbers and mtbody is a function such that

```
                    all m n.(mtbody(m,n) is a scalar)
          all m n.(column < m or row < n imp mtbody(m,n)=0).
```

3.2. Files

The proof consists of the following three files.

NAINUM.LSP
: axioms and theorems on natural numbers by Ketonen

SUM.LSP
: axioms on scalars and theorems on summation

MAT.LSP
: the proof of associativity

4. Quick Sort

4.1. Type and Sorts

Since EKL does not allow inductive definition of a domain, the set of lists, on which the quick sort algorithm operates, is represented by sort `list` of type `ground:truthval`. `element`, the set of elements of lists, is also a sort of type `ground:truthval` and is not specified explicitly; i.e. it may be replaced by any sort on which the total order is defined.

The total order on `element` is realized by the binary predicate `leq` and its properties expressed as axioms such as the one below.

```
(axiom "all a b.leq(a,b) or leq(b,a)")
(label totalorder)
```

Lists are thought to be constructed by constructors `ku` and `cons`.

```
(decl ku (type ground) (syntype constant) (sort list))
(decl cons (type "(ground,ground):ground") (syntype constant))
```

`ku` is the empty list (`ku` means empty in Japanese). `cons`, given an element and a list, returns another list; this is expressed by the following axiom.

```
(axiom "all a x.list(cons(a,x))")
```

Note that a is a variable of type element and x is a variable of type list.

element and list are the only sorts in the proof; we do not use natnum or any other sorts. It means that the result of the proof is weak in some sense; see Section 4 of this chapter.

4.2. Induction

Since we do not use natnum, the induction used in the proof are all on lists. The simple induction schema on lists is given as an axiom.

```
(axiom "all p.(p(ku) & all a x.p(x) imp p(cons(a,x))) imp
              all x.p(x)")
```

The course-of-values induction schema, which we call shorter induction (labeled with shorterind), is expressed in terms of a binary predicate shorter.

```
all p.(all x.(all y.shorter(y,x) imp p(y)) imp p(x)) imp
             all x.p(x)
```

shorter(x,y) means that list x is strictly shorter than list y; shorter is defined as follows.

```
(defax shorter "all x y.shorter(x,y) iff
                    x = ku & y != ku or
                    x != ku & y != ku &
                    shorter(tail(x),tail(y))")
```

The validity of the course-of-values induction schema is proved by the simple induction schema.

4.3. Function Definition

The functions realizing the quick sort algorithm, i.e. qsort, partl, partr, and append, are all defined by an equation with the EKL DEFAX form.

The fact that a function terminates with some input value is interpreted by the fact that the value of the function lies in list; e.g. the totality of qsort is expressed by the following formula.

```
all x.list(qsort(x))
```

The partition is realized by two functions partl (partition left) and partr (partition

right); if we define the partition as a function returning two values, the proof will be full of projections extracting the result of the partition.

4.4. What has been Proved

The main theorems of the proof are:

```
all x a.member(a,qsort(x)) imp member(a,x)
all x a.member(a,x) imp member(a,qsort(x))
all x.sorted(qsort(x))
```

The first two say that each element in the input list is an element of the sorted list and vice versa. They do not include the notion of occurrence of an element in a list. If, however, we define the occurrence of an element in a list by a function, say `occ`, whose value will be a natural number, and prove the appropriate properties of `occ`, then the proof can be easily extended to include the desired properties of `occ` and `qsort`; especially, the pattern of the induction will be completely identical.

The third theorem is what we wanted most. The function `sorted` is defined as follows.

```
(defax sorted
       "all x.sorted(x) iff
              (x = ku or
               x != ku & tail(x) = ku or
               x != ku & tail(x) != ku &
               leq(head(x),head(tail(x))) &
               sorted(tail(x)))")
```

4.5. Rewriter

We do not use the label `simpinfo`. Instead, the fundamental axioms (or lemmas) on lists are labeled with `listinfo`, and in most of rewriting, we use the rewriter `(use listinfo)`.

4.6. Files

The proof consists of the following files, which will be listed in the order below.

QSORT.LSP
: basic definitions

SHORTER_LEMMAS.LSP
: lemmas on shorter

SHORTERIND.LSP
: proof of shorter induction

MEMBER_APPEND_LEMMA.LSP
lemma on member and append
SORTED_MEMBER_APPEND.LSP
lemma on sorted, member and append
PARTITION_LEMMAS.LSP
lemmas on partition
MACROS.LSP
some macros used below
QSORT_IS_TOTAL.LSP
proof of totality of qsort
QSORT_SORTS.LSP
main theorems

5. Miscellaneous Experiments

5.1. Rules on Derivatives in Synthetic Differential Geometry

Synthetic differential geometry (SDG) is one of the target theories of CAP project. EKL is not really suitable for SDG, since SDG uses intuitionistic logic. But this does not affect the following experiment. For the detail of SDG, see [Kock 81]. The proofs is in DIFF.LSP.

5.2. Non-standard Analysis

Ballantyne and Bledsoe have proved the theorem that any continuous function on a compact set is uniformly continuous on it by their stand-alone theorem prover based on non-standard analysis [cf. Ballantyne & Bledsoe 77]. We proved the same theorem by the same method in EKL. The proof is in BB.LSP.

References

Ballantyne, A.M. & Bledsoe, W.W. [1977]: Automatic Proofs of Theorems in Analysis Using Non-standard Techniques, JACM 24, 353-374

Furukawa, K. & Yokoi, T. [1984]: Basic Software System, in Proceedings of the International Conference on Fifth Generation Computer Systems 1984, 37-57

Ketonen, J. & Weening, J.S. [1984]: EKL - An Interactive Proof Checker User's Manual, Stanford University

Ketonen, J. & Weening, J.S. [1984a]: The Language of an Interactive Proof Checker, Stanford University

Kock, A. [1981]: Synthetic Differntial Geometry, London Mathematical Society Lecture Note

Series 51. Cambridge University Press.

Listings

```
(wipe-out)
(proof natnum)

(decl lessp (type !(ground,ground):truthval!) (syntype constant)
     (infixname <) (bindingpower 920))

(decl add1 (type !ground:ground!) (syntype constant) (postfixname !'!)
     (bindingpower 975))

(decl plus (infixname +) (type !(ground,ground,(ground*)):ground!)
     (syntype constant) (associativity both) (bindingpower 930))

(decl times (type !(ground,ground,(ground*)):ground!)
     (syntype constant) (infixname *) (associativity both)
     (bindingpower 935))

(decl (i j k n m) (sort !natnum!) (type !ground!))

(decl (a b c set) (type !ground:truthval!))

;needed axioms on order

(axiom !all n.not n < n!)
(label irreflexivity_of_order)

(axiom !all n m k.n < m&m < k imp n < k!)
(label transitivity_of_order)

(axiom !all n.not n < 0!)
(label zeroleast1)

;successor and order

(axiom !all n.natnum(n')!)
(label simpinfo)

(axiom !all n.n < n'!)
(label successor1)(label succfacts)

(axiom !all n m.not n < m imp m < n'!)
(label successors2)(label succfacts)

(axiom !all n m.n' < m' iff n < m!)
(label successorless)(label succfacts)

(axiom !all n m.(n' = m') iff (n = m)!)
(label successoreq)(label succfacts)

(axiom !all n.not n = 0 imp 0 < n!)
(label zeroleast2)(label succfacts)

(axiom !all n.0 < n'!)
(label zeroleast3)(label succfacts)

(axiom !all n.not(n' = 0)!)
(label zero_not_successor)(label succfacts)

;definition of predecessor

(decl pred (type !ground:ground!) (syntype constant))
(defax pred !all n.pred(n') = n!)
```

```
(axiom !all n.natnum pred n!)
(label simpinfo)

;addition

(defax plus !all n k.0+n=n & k'+n=(k+n)'!)
(label simpinfo)(label plusfacts)

(axiom !all n m.natnum(n+m)!)
(label simpinfo)

(axiom !all n.n+0 =n!)
(label simpinfo)(label plusfacts)

(axiom !all n. 1+n=n' & n+1=n'!)
(label simpinfo)(label plusfacts)

(axiom !all n k.n+k'=(n+k)'!)
(label simpinfo)(label plusfacts)

(axiom !all n k m.(k+m=k+n) iff (m=n)!)
(label lpluscan)(label plusfacts)

(axiom !all n k m.(m+k=n+k) iff (m=n)!)
(label rpluscan)(label plusfacts)

(axiom !all n k.n+k=0 iff n=0 & k=0!)
(label addtozero)(label plusfacts)

(axiom !all k n.k+n=n+k!)
(label commutadd)(label plusfacts)

;multiplication

(defax times !all n k.0*n=0 & n'*k=(n*k)+k!)
(label simpinfo)(label timesfacts)

(axiom !all n m.natnum(m*n)!)
(label simpinfo)

(axiom !all n.n*0=0 & 1*n=n & n*1=n!)
(label simpinfo)(label timesfacts)

(axiom !all n k.n*k'=n*k+n!)
(label timsucc)(label timesfacts)

(axiom !all n k m.not k=0 imp ((k*m=k*n) iff (m=n))!)
(label ltimescan)(label timesfacts)

(axiom !all n k m.not k=0 imp ((m*k=n*k) iff (m=n))!)
(label rtimescan)(label timesfacts)

(axiom !all n m.n*m=m*n!)
(label commutmult)(label timesfacts)

(axiom !all n k.not n=0 imp n*k=0 iff k=0!)
(label ltimestozero)(label timesfacts)

(axiom !all n k.not n=0 imp k*n=0 iff k=0!)
(label rtimestozero)(label timesfacts)
```

```
;distributivity

(axiom !all n k m.n*(k+m)=k*n+m*n!)
(label idistrib) (label timesfacts) (label plusfacts)

(axiom !all n m k.(m+k)*n=m*n+k*n!)
(label rdistrib) (label timesfacts) (label plusfacts)

;inductive principles

(proof induction)

(axiom !all a.a(0)&(all n.a(n) imp a(n')) imp (all n.a(n))!)
(label proof_by_induction)

(decl npars (type !ground*!))
(decl ndf (type !(ground,ground*):ground*!))
(decl zcase (type !ground*:ground!))
(axiom
 !all ndf zcase ndef.
   (ex fun.
     (all npars n.fun(0,npars)=zcase(npars)&
                  fun(n',npars)=ndef(n,fun(n,ndf(n,npars)),npars)))!)
(label inductive_definition)

;the following is a form of double induction

(axiom !all a2.(all n m.a2(n,0)&(a2(n,m) imp a2(n',m'))) imp all n m.a2(n,m)!)
(label proof_by_doubleinduction)

;(save-proofs natnum)
```

```
;; scalar and its finite summation

(load "udd:ekl:prf:natnum.lsp")
(proof sum)

(decl scalar (syntype constant) (type !ground:truthval!))

(decl scalar_plus (type !(ground,ground,ground*):ground!)
      (syntype constant) (infixname ++)
      (bindingpower 930) (associativity both))

(decl scalar_times (type !(ground,ground,ground*):ground!)
      (syntype constant) (infixname **)
      (bindingpower 935) (associativity both))

(decl (szero sunit) (syntype constant) (type !ground!) (sort !scalar!))

(decl (aa bb cc) (sort !scalar!) (type !ground!))

(axiom !all aa bb.scalar(aa ++ bb)!)
(label simpinfo)(label scalar_plus1)

(axiom !all aa bb.scalar(aa ** bb)!)
(label simpinfo)

(axiom !all aa.( aa ++ szero = aa &  szero ++ aa = aa)!)
(label simpinfo)

(axiom !all aa.( aa ** sunit = aa &  sunit ** aa = aa)!)
(label simpinfo)

(decl (ff gg hh) (type !ground:ground!))

(decl series (syntype constant) (type !(ground:ground):truthval!))

(define series !series=lambda ff. all i.scalar(ff(i))!)

(decl (f g h) (type !ground:ground!) (sort !series!))

(decl sum (syntype constant) (type !(ground,(ground:ground)):ground!))

(defax sum
 !all ff.(sum(0,ff)=szero) & all i ff.(sum(i',ff)=(ff(i') ++ sum(i,ff)))!)
(label simpinfo)(label sumfacts)(label sumdef)

(trw !all f. series(f)!)

(rw * (open series))
(label simpinfo)
:all f i.scalar(f(i))

(trw !all i f.(scalar(sum(i,f)) imp scalar(sum(i',f)))! (use sumdef mode always) (use
:all i f.scalar(sum(i,f)) imp scalar(sum(i',f))

(ue ((a . !lambda i.scalar(sum(i,f))!)) proof_by_induction (use *))
:all n.scalar(sum(n,f))

(derive !all n f.scalar(sum(n,f))! *)
(label simpinfo)

:simpfacts
```

```
(axiom |szero != sunit|)
(label simpinfo)

;smultiplication

(axiom |all aa. aa ** szero = szero & szero ** aa = szero|)
(label simpinfo)(label smultfacts)

(axiom |all aa. aa ** sunit = aa & sunit ** aa = aa|)
(label simpinfo)(label smultfacts)

(axiom |all aa bb. aa ** bb = bb ** aa|)
(label simpinfo)(label smultfacts)(label scalar_product_commutes)


;sdivide

(decl sdivide (syntype constant) (type |(ground,ground):ground|)
      (infixname %%) (bindingpower  936))

(axiom |all aa bb. aa != szero imp scalar(bb %% aa)|)
(label simpfino)(label sdividefacts)

(axiom |all aa bb.(aa != szero imp (aa ** bb %% aa) = bb)|)
(label simpinfo)(label sdividefacts)

;saddition

(axiom |all aa. aa ++ szero = aa|)
(label simpinfo)(label splusfacts)

(axiom |all aa bb. aa ++ bb = bb ++ aa|)
(label simpfino)(label splusfacts)(label scalar_plus_commutes)

;sminus

(decl sminusu (syntype constant) (type |ground:ground|)
      (prefixname -) (bindingpower 937))

(decl sminusb (syntype constant) (type |(ground,ground):ground|)
      (infixname --) (bindingpower 937))

(axiom |all aa. scalar(- aa)|)
(label simpfino)(label sminusfacts)

(axiom |all aa. aa ++ - aa = szero|)
(label simpfino)(label sminusfacts)

(axiom |all aa bb. scalar(aa -- bb)|)
(label simpinfo)

(axiom |all aa bb. (aa -- bb) ++ bb = aa|)
(label simpinfo)



;distributivity
(axiom |all aa bb cc. aa ** (bb ++ cc) = aa ** bb ++ aa ** cc|)
(label simpinfo)(label distfacts)
```

```
(axiom !all aa bb cc. (aa ++ bb) ** cc = aa ** cc ++ bb ** cc!)
(label simpinfo)(label distfacts)



;; distributivity of scalar product and sum

;scalar product of series
(decl sps (type !(ground,(ground:ground)):(ground:ground)!)
          (syntype constant))

(decl (xx yy zz) (type !ground!))

(define sps !all xx ff. sps(xx,ff)= (lambda i. xx ** ff(i))!)
(label spsdef)

(axiom !all aa f. series(sps(aa,f))!)
(label simpinfo)

:PROOF
(ue ((a . !lambda i.(aa ** sum(i,f) = sum(i,sps(aa,f)))!))
    proof_by_induction (use sumfacts mode always) (use spsdef)
    (use distfacts mode exact))

(derive !all f i.(aa ** sum(i,f) = sum(i,sps(aa,f)))! *)
(label sum_prop1)
:QED

(rw sum_prop1 (open sps))
(label sum_prop1_1)

(trw !all f i.sum(i,f) ** aa = sum(i,lambda i1.f(i1) ** aa)! *)
(label sum_prop1_2)

;; commutativity of sum and ++

;addition of series
(decl series_plus (type !((ground:ground),(ground:ground)):(ground:ground)!)
      (syntype constant))

(define series_plus !all ff gg. series_plus(ff,gg)=lambda i.ff(i) ++ gg(i)!)
(label series_plusdef)

(axiom !all f g.series(series_plus(f,g))!)
(label simpinfo)

:PROOF
(assume !sum(n,series_plus(f,g)) = sum(n,f) ++ sum(n,g)!)

(trw !sum(n',series_plus(f,g))=sum(n',f) ++ sum(n',g)!
     (use sumfacts mode exact) (use * mode exact)
     (use series_plusdef mode exact)
     (use scalar_plus_commutes direction simpler))
;sum(n',series_plus(f,g))=sum(n',f) ++ sum(n',g)

(ci "-2" *)

(ue ((a . !lambda i.sum(i,series_plus(f,g)) = sum(i,f) ++ sum(i,g)!))
   proof_by_induction (use * mode exact))

(derive !all f g i.sum(i,series_plus(f,g)) = sum(i,f) ++ sum(i,g)! *)
```

```
(label simpinfo) (label sum_prop2)
:QED


:double sereis

(decl dblseries (syntype constant) (type !((ground,ground):ground):truthval!))

(decl bff (type !(ground,ground):ground!))

(decl (i j k l m n) (type ground) (sort natnum))

(define dblseries !dblseries=lambda bff. all i j.scalar(bff(i,j))!)

(decl (d1 d2 d3) (type !(ground,ground):ground!) (sort dblseries))

(trw !all d1. dblseries(d1)!)
(label simpinfo)

(rw * (open dblseries) (nuse simpinfo))
(label simpinfo)

(trw !all d1 i. series(lambda k. d1(k,i)) & series(lambda k. d1(i,k))!
     (open series) (use *))
(label simpinfo)

;; commutativity of summation

;LEMMA
(ue ((a . !lambda j0.(szero = sum(j0,lambda j.szero))!))
    proof_by_induction (use sumdef mode exact))
;all n.szero = sum(n,lambda j.szero)
(label sum_prop3_hlp0)
:QED

:PROOF
(trw !series(lambda j. sum(n,lambda k. d1(k,j)))! (open series))
;series(lambda j. sum(n,lambda k. d1(k,j)))

(ue ((f . !lambda j. d1(n',j)!) (g . !lambda j. sum(n,lambda k. d1(k,j))!))
    sum_prop2
    (nuse simpinfo) ; without this option, EKL rewrites this to true
    (use *)
    (open series_plus))
;all i.sum(i,lambda i1.d1(n',i1) ++ sum(n,lambda k.d1(k,i1))) =
;       sum(i,lambda j.d1(n',j)) ++
;       sum(i,lambda j.sum(n,lambda k.d1(k,j)))
(label sum_prop3_hlp1)

(ue ((a . !lambda k0.sum(k0,lambda k.sum(j0,lambda j.d1(k,j)))
            = sum(j0,lambda j. sum(k0,lambda k. d1(k,j)))!))
    proof_by_induction
    (use sum_prop3_hlp0 mode exact)
    (use sum_prop3_hlp1 mode exact)
    (use sumfacts mode exact))
;all n.sum(n,lambda k.sum(j0,lambda j.d1(k,j))) =
;       sum(j0,lambda j.sum(n,lambda k.d1(k,j)))

(derive !all n.sum(n,lambda k.sum(j0,lambda j.d1(k,j))) =
           sum(j0,lambda j.sum(n,lambda k.d1(k,j)))! *)
(label sum_prop3)
```

:QED

```
;; definition of matrix and its product

(load "tool")
(eload ":udd:ekl:prf:sum.lsp")
(proof mat)


;product of two double series as matrix body
(decl mtpr
      (syntype constant)
      (type !(ground,((ground,ground):ground),((ground,ground):ground)):
            ((ground,ground):ground)!))

(define mtpr !mtpr=
       lambda j0 d1 d2.lambda i k. sum(j0,lambda j,d1(i,j) ** d2(j,k))!)
(label mtprdef)

(trw !all i k d1 d2. series(lambda j,d1(i,j) ** d2(j,k))! (open series))
(label simpinfo)

;LEMMATTA
(trw
 !all j0 d1 d2. dblseries(lambda i k.sum(j0,lambda j,d1(i,j) ** d2(j,k)))!
     (open dblseries))
(label simpinfo)

(trw !dblseries(lambda m n .(d1(i,m) ** d2(m,n) ** d3(n,k)))!
     (open dblseries))
(label simpinfo)
;QED

;MATRIX PROPOSITION 1 (associativity of mtpr)

;PROOF
(trw
 !mtpr(k0,d1,mtpr(j0,d2,d3)) = mtpr(j0,mtpr(k0,d1,d2),d3)!
 (nuse scalar_product_commutes)
 (open mtpr)
 (use sum_prop1_1 mode exact)
 (use sum_prop1_2 mode exact)
 (use sum_prop3 ue ((d1 . !lambda m n . d1(i,m) ** d2(m,n) ** d3(n,k)!))))

(derive
   !all k0 j0 d1 d2 d3.
        mtpr(k0,d1,mtpr(j0,d2,d3)) = mtpr(j0,mtpr(k0,d1,d2),d3)! *)
(label mat_prop1)
;QED


(decl bff (type !(ground,ground):ground!))

(decl matrix (syntype constant)
      (type !(ground,ground,((ground,ground):ground)):truthval!))

(decl (grnd grnd1 grnd2) (type ground))

(decl (mm1 mm2) (type !(ground,ground,((ground,ground):ground))!))

;column row and matrix body
(decl column
     (syntype constant)
```

```
    (type !(ground,ground,((ground,ground):ground)):ground!))

(decl row
     (syntype constant)
    (type !(ground,ground,((ground,ground):ground)):ground!))

(decl mtbody
      (syntype constant)
      (type
       !(ground,ground,((ground,ground):ground)):((ground,ground):ground)!))

(define column !column = lambda grnd1 grnd2 bff. grnd1!)
(label columndef)

(define row !row = lambda grnd1 grnd2 bff. grnd2!)
(label rowdef)

(define mtbody !mtbody = lambda grnd1 grnd2 bff. bff!)
(label mtbodydef)

(name-rewriter mtacs (use columndef rowdef mtbodydef mode exact))

(name-rewriter mtacsrev (use columndef rowdef mtbodydef direction reverse))

(define matrix
  !matrix
      =
   lambda mm1
     (natnum(column(mm1)) & natnum(row(mm1)) & dblseries(mtbody(mm1)) &
       all m n.((column(mm1) < m or row(mm1) < n) imp (mtbody(mm1))(m,n)=0))!)
(label matrixdef)

(decl matrix_product
      (syntype constant)
      (type
         !((ground,ground,((ground,ground):ground)),
           (ground,ground,((ground,ground):ground)))
           :(ground,ground,((ground,ground):ground))!)
      (infixname ***) (bindingpower 935))

(decl (aaa bbb ccc)
      (type !(ground,ground,((ground,ground):ground))!)
      (sort matrix))

(decl max (syntype constant) (type !(ground,ground):ground!))

(define max !max=lambda i j. if i < j then j else i!)
(label maxdef)

(define matrix_product
   !matrix_product =
     lambda mm1 mm2
        (column mm1,
         row mm2,
         mtpr(max(row mm1,column mm2),mtbody mm1,mtbody mm2))!)
(label matrix_productdef)

;; associativity of matrix_product (***)


(decl (aaa bbb ccc) (type !(ground,ground,((ground,ground):ground))!)
```

```
      (sort matrix))

:LEMMATTA
(trw !all aaa. matrix(aaa)!)

(rw * (open matrix))
!all aaa.natnum(column(aaa))&natnum(row(aaa))&dblseries(mtbody(aaa))&
!        (all m n.column(aaa) < m or row(aaa) < n imp
!                 (mtbody(aaa))(m,n) = szero)

(trw
 !all aaa. natnum(column(aaa)) & natnum(row(aaa)) & dblseries(mtbody(aaa))! *)
(label simpinfo)

(trw !all m n.natnum(max(m,n))! (open max))
(label simpinfo)

(trw !all aaa bbb.
     (natnum(max(column(aaa),row(bbb)))
             &
       natnum(max(row(aaa),column(bbb))))!)
(label mat_prop2_hlp1)
:QED

; MATRI\ THEOREM 1

:PROOF
(trw !aaa *** (bbb *** ccc) = (aaa *** bbb) *** ccc!
     (open matrix_product) (use sum_prop1 mode exact)
     (use sum_prop2 mode exact) (use sumdef mode exact)
     @mtacs)
!the returned formula is too huge to display

(rw *
    @mtacsrev
    (use mat_prop1 mode exact)
    (use mat_prop2_hlp1))
!aaa *** (bbb *** ccc) = (aaa *** bbb) *** ccc

(derive !all aaa bbb ccc.aaa *** (bbb *** ccc) = (aaa *** bbb) *** ccc! *)
(label mat_th1)
:QED
```

```
;;;;    QSORT.LSP
;;;;
;;;;    Basic file for proving properties of QSORT (quick-sort)

(wipe-out)
(proof qsort)

(decl element (type "ground:truthval") (syntype constant))
(decl leq (type "(ground,ground):truthval") (syntype constant))
(decl (a b c) (type ground) (sort element))

(axiom "all a.leq(a,a)")
(label order)
(axiom "all a b c.leq(a,b) & leq(b,c) imp leq(a,c)")
(label order)
(axiom "all a b.leq(a,b) & leq(b,a) imp a = b")
(label order)
(axiom "all a b.leq(a,b) or leq(b,a)")
(label totalorder)
(label order)

(axiom "element(1) & element(2) & element(3)")
(label small_model)
(axiom "leq(1,1) & leq(1,2) & leq(1,3) & leq(2,2) & leq(2,3) & leq(3,3)")
(label small_model)
(axiom "not leq(2,1) & not leq(3,1) & not leq(3,2)")
(label small_model)

(decl list (type "ground:truthval") (syntype constant))
(decl ku (type ground) (syntype constant) (sort list))
(decl cons (type "(ground,ground):ground") (syntype constant))
(decl head (type "ground:ground") (syntype constant))
(decl tail (type "ground:ground") (syntype constant))
(decl append (type "(ground,ground):ground") (syntype constant))

(decl shorter (type "(ground,ground):truthval") (syntype constant))

(decl member (type "(ground,ground):truthval") (syntype constant))
(decl sorted (type "ground:truthval") (syntype constant))
(decl (x y z) (type ground) (sort list))

(axiom "all p.(p(ku) & all a x.p(x) imp p(cons(a,x))) imp all x.p(x)")
(label listind)

(axiom "all a x.list(cons(a,x))")
(label listinfo)

(axiom "all a x.cons(a,x) != ku")
(label listinfo)

(axiom "all x.x != ku imp element(head(x))")
(label listinfo)

(axiom "all x.x != ku imp list(tail(x))")
(label listinfo)

(axiom "all a x.head(cons(a,x)) = a")
(label listinfo)

(axiom "all a x.tail(cons(a,x)) = x")
(label listinfo)
```

```
(axiom "all x,x != ku imp cons(head(x),tail(x)) = x")
(label listinfo)

(defax append
       "all x y,append(x,y) =
                if x = ku then y
                   else cons(head(x),append(tail(x),y))")
(label appenddef)


;Theorem (append_is_total)
;all x y,list(append(x,y))

(ue ((p . "lambda x,all y,list(append(x,y))"))
    listind
    (part ! (use listinfo) (open append)))
(label listinfo)


(defax shorter "all x y,shorter(x,y) iff
                          x = ku & y != ku or
                          x != ku & y != ku & shorter(tail(x),tail(y))")

(axiom "all x y a,shorter(x,y) imp shorter(x,cons(a,y))")
(label shorter_lemma1)
;This lemma is proved in SHORTER_LEMMAS.LSP.

(axiom "all a x,shorter(x,cons(a,x))")
(label shorter_lemma2)
;This lemma is proved in SHORTER_LEMMAS.LSP.

(axiom "all x y z a,shorter(y,cons(a,x)) & shorter(z,y) imp shorter(z,x)")
(label shorter_lemma3)
;This lemma is proved in SHORTER_LEMMAS.LSP.


(axiom "all p,(all x,(all y,shorter(y,x) imp p(y)) imp p(x)) imp
              all x,p(x)")
(label shorterind)
;This theorem is proved in SHORTERIND.LSP.


(defax member
       "all a x,member(a,x) iff
                x != ku &
                (head(x) = a or member(a,tail(x)))")


(axiom "all a x y b,member(b,append(x,cons(a,y))) imp
                    member(b,x) or b = a or member(b,y)")
(label member_append_lemma_if)
;This lemma is proved in MEMBER_APPEND_LEMMA.LSP.

(axiom "all a x y b,member(b,x) or b = a or member(b,y) imp
                    member(b,append(x,cons(a,y)))")
(label member_append_lemma_only_if)
;This lemma is proved in MEMBER_APPEND_LEMMA.LSP.


(defax sorted
```

```
       "all x.sorted(x) iff
              (x = ku or
               x != ku & tail(x) = ku or
               x != ku & tail(x) != ku & leq(head(x),head(tail(x))) &
               sorted(tail(x)))")

(axiom "all a x y.sorted(x) & sorted(y) &
                  (all b.member(b,x) imp leq(b,a)) &
                  (all b.member(b,y) imp not leq(b,a)) imp
                  sorted(append(x,cons(a,y)))")
(label sorted_member_append)
;This theorem is proved in SORTED_MEMBER_APPEND.LSP.


;Definitions (quick-sort and partition)

(decl partl (type "(ground,ground):ground") (syntype constant))
(decl partr (type "(ground,ground):ground") (syntype constant))
(decl qsort (type "ground:ground") (syntype constant))

(defax partl
       "all a x.partl(a,x) =
                if x = ku then ku
                   else (if leq(head(x),a)
                            then cons(head(x),partl(a,tail(x)))
                            else partl(a,tail(x)))")
(label partldef)

(defax partr
       "all a x.partr(a,x) =
                if x = ku then ku
                   else (if leq(head(x),a)
                            then partr(a,tail(x))
                            else cons(head(x),partr(a,tail(x))))")
(label partrdef)

(defax qsort
       "all x.qsort(x) =
              if x = ku or tail(x) = ku then x
                 else append(qsort(partl(head(x),tail(x))),
                             cons(head(x),qsort(partr(head(x),tail(x)))))")
(label qsortdef)


;Theorem (partl_is_total):
;all a x.list(partl(a,x))

(ue ((p . "lambda x.all a.list(partl(a,x))"))
    listind
    (part 1 (use listinfo) (open partl)))
(label listinfo)


;Theorem (partr_is_total):
;all a x.list(partl(a,x))

(ue ((p . "lambda x.all a.list(partr(a,x))"))
    listind
    (part 1 (use listinfo) (open partr)))
(label listinfo)
```

```
;(trw "qsort(cons(2,(cons(1,cons(3, ku)))))"
;     (use listinfo)
;     (use small_model)
;     (use appenddef partldef partrdef qsortdef mode always))
```

```
;;;;    SHORTER_LEMMAS.LSP

(proof shorter_lemmas)


;Theorem (shorter_lemma1)
;all x y a.shorter(x,y) imp shorter(x,cons(a,y))

(assume "all y a.shorter(x,y) imp shorter(x,cons(a,y))")
(label shorter_lemma1_ih)

(assume "shorter(cons(b,x),y)")
(label shorter_lemma1_premise)

(rw * (use listinfo) (open shorter))

(ue ((y . "tail(y)") (a . "head(y)")) shorter_lemma1_ih
    (use listinfo) (use *))

(trw "shorter(cons(b,x),cons(a,y))" (use listinfo) (open shorter) (use *))

(ci shorter_lemma1_premise *)

(derive "all y a.shorter(cons(b,x),y) imp shorter(cons(b,x),cons(a,y))" *)

(ci shorter_lemma1_ih *)

(ue ((p . "lambda x.all y a.shorter(x,y) imp shorter(x,cons(a,y))"))
    listind
    (use listinfo) (part 1 (part 1 (open shorter)))
    (use *))


;Theorem (shorter_lemma2)
;all a x.shorter(x,cons(a,x))

(ue ((p . "lambda x.all a.shorter(x,cons(a,x))")) listind
    (use listinfo) (part 1 (open shorter)))


;Theorem (shorter_lemma3)
;all x y z a.shorter(y,cons(a,x)) & shorter(z,y) imp shorter(z,x)")

(assume "shorter(y,cons(a,x)) & shorter(ku,y)")
(rw * (use listinfo) (open shorter))
(rw * (use listinfo) (open shorter))
(derive "x != ku" *)
(trw "shorter(ku,x)" (use listinfo) (open shorter) (use *))
(ci -5 *)
(label shorter_lemma3_base)

(assume "all x y a.shorter(y,cons(a,x)) & shorter(z,y) imp shorter(z,x)")
(label shorter_lemma3_ih)

(assume "shorter(y,cons(a,x)) & shorter(cons(b,z),y)")
(label shorter_lemma3_premise)

(rw * (use listinfo) (part 1 (open shorter)) (part 2 (open shorter)))

(assume "x != ku")
```

```
(ue ((x . "tail(x)") (y . "tail(y)") (a . "head(x)")) shorter_lemma3_ih
    (use listinfo) (use -2 *))

(trw "shorter(cons(b,z),x)" (use listinfo) (open shorter) (use -2 *))
(label shorter_lemma3_case1)

(assume "x = ku")

(rw shorter_lemma3_premise
    (use listinfo) (use *)
    (part 1 (open shorter)))
(rw * (use listinfo) (part 1 (open shorter)))
(rw * (use listinfo) (open shorter))
(derive "false imp shorter(cons(b,z),x)")
(derive "shorter(cons(b,z),x)" (-2 *))
(label shorter_lemma3_case2)

(derive "x != ku or x = ku")
(cases * shorter_lemma3_case1 shorter_lemma3_case2)

(ci shorter_lemma3_premise *)

(derive "all x y a.shorter(y,cons(a,x)) & shorter(cons(b,z),y) imp
                    shorter(cons(b,z),x)"
        *)

(ci shorter_lemma3_ih *)

(ue ((p . "lambda z.all x y a.shorter(y,cons(a,x)) & shorter(z,y) imp
                              shorter(z,x)"))
    listind
    (use listinfo)
    (use shorter_lemma3_base) (use *))
```

```
;;;;    SHORTERIND.LSP

(proof shorterind)

;Theorem (shorterind):
;all p.(all x.(all y.shorter(y,x) imp p(y)) imp p(x)) imp
;       all x.p(x)"


(assume "all x.(all y.shorter(y,x) imp p(y)) imp p(x)")
(label shorterind_assumption)

(derive "all y.shorter(y,ku) imp p(y)" nil (use listinfo) (open shorter))
(label shorterind_base)

(assume "all y.shorter(y,x) imp p(y)")
(label shorterind_ih)

(assume "shorter(y,cons(a,x))")
(label shorterind_premise)

(assume "shorter(z,y)")
(ue ((x . x) (y . y) (z . z) (a . a)) shorter_lemma3 (use -2 -1))
(ue ((y . z)) shorterind_ih (use *))
(ci -3 *)
(ue ((x . y)) shorterind_assumption (use *))
(ci shorterind_premise *)
(derive "all y.shorter(y,cons(a,x)) imp p(y)" *)
(ci shorterind_ih *)

(ue ((p . "lambda x.all y.shorter(y,x) imp p(y)")) listind
    (use shorterind_base *) (use listinfo))

(ue ((a . a) (x . x)) shorter_lemma2)
(ue ((x . "cons(a,x)") (y . x)) -2 (use *) (use listinfo))

(derive "all x.p(x)" *)

(ci shorterind_assumption *)

(define shorterind_temp
        "shorterind_temp =
         lambda p.(all x.(all y.shorter(y,x) imp p(y)) imp p(x)) imp
                   all x.p(x)")

(trw "shorterind_temp(p)" (use -2) (open shorterind_temp))

(derive "all p.shorterind_temp(p)" *)

(rw * (open shorterind_temp))
```

```
;;;;    MEMBER_APPEND_LEMMA.LSP

(proof member_append_lemma)


;Theorem (member_append_lemma_if)
;lambda x,all a y b.member(b,append(x,cons(a,y))) imp
;                    member(b,x) or b = a or member(b,y)

(assume "all a y b.member(b,append(x,cons(a,y))) imp
                   member(b,x) or b = a or member(b,y)")
(label member_append_lemma_if_ih)

(assume "member(b,append(cons(c,x),cons(a,y)))")
(label member_append_lemma_if_premise)

(rw * (use listinfo) (open append) (open member))

(derive "b = c or member(b,x) or b = a or member(b,y)"
        (* member_append_lemma_if_ih))

(derive "member(b,cons(c,x)) or b = a or member(b,y)" *
        (use listinfo) (part 1 (open member)))

(ci member_append_lemma_if_premise *)

(derive "all a y b.member(b,append(cons(c,x),cons(a,y))) imp
                   member(b,cons(c,x)) or b = a or member(b,y)"
        *)

(ci member_append_lemma_if_ih *)

(ue ((p . "lambda x.all a y b.member(b,append(x,cons(a,y))) imp
                              member(b,x) or b = a or member(b,y)"))
    listind
    (use listinfo) (part 1 (part 1 (open append) (open member)))
    (use *))


;Theorem (member_append_lemma_only_if)
;lambda x.all a y b.member(b,x) or b = a or member(b,y) imp
;                    member(b,append(x,cons(a,y)))

(assume "all a y b.member(b,x) or b = a or member(b,y) imp
                   member(b,append(x,cons(a,y)))")
(label member_append_lemma_only_if_ih)

(assume "member(b,cons(c,x)) or b = a or member(b,y)")
(label member_append_lemma_only_if_premise)

(rw * (use listinfo) (part 1 (open member)))

(derive "b = c or member(b,append(x,cons(a,y)))"
        (* member_append_lemma_only_if_ih))

(derive "member(b,append(cons(c,x),cons(a,y)))" *
        (use listinfo) (open append) (open member))

(ci member_append_lemma_only_if_premise *)

(derive "all a y b.member(b,cons(c,x)) or b = a or member(b,y) imp
```

```
                    member(b,append(cons(c,x),cons(a,y)))"
        *)

(ci member_append_lemma_only_if_ih *)

(ue ((p . "lambda x,all a y b,member(b,x) or b = a or member(b,y) imp
                                member(b,append(x,cons(a,y)))"))
    listind
    (use listinfo)
    (part 1 (part 1 (part 1 (part 1 (part 1 (open member))))))
    (part 1 (part 1 (part 1 (part 2 (open append) (open member)))))
    (use *))
```

```
;;;;    SORTED_MEMBER_APPEND.LSP


(proof sorted_member_append)

;Theorem (sorted_member_append):
;all x a y.sorted(x) & sorted(y) &
;          (all b.member(b,x) imp leq(b,a)) &
;          (all b.member(b,y) imp not leq(b,a)) imp
;          sorted(append(x,cons(a,y)))


(assume "sorted(y)")
(label sorted_member_append_base_premise1)

(assume "all b.member(b,y) imp not leq(b,a)")
(label sorted_member_append_base_premise2)

(assume "y = ku")
(rw "sorted(append(ku,cons(a,y)))"
     (use *) (use listinfo) (open append) (open sorted))
(label sorted_member_append_base_case1)

(assume "not y = ku")
(ue ((b . "head(y)")) sorted_member_append_base_premise2
    (use *) (use listinfo) (open member))
(ue ((a . "a") (b . "head(y)")) totalorder (use -2 *) (use listinfo))
(rw "sorted(append(ku, cons(a,y)))"
     (use -3 * sorted_member_append_base_premise1) (use listinfo)
     (open append) (open sorted))
(label sorted_member_append_base_case2)

(derive "y = ku or not y = ku")
(cases * sorted_member_append_base_case1 sorted_member_append_base_case2)

(ci (sorted_member_append_base_premise1 sorted_member_append_base_premise2)
    *)
(label sorted_member_append_base)

(assume "all a y.sorted(x) & sorted(y) &
                 (all b.member(b,x) imp leq(b,a)) &
                 (all b.member(b,y) imp not leq(b,a)) imp
                 sorted(append(x,cons(a,y)))")
(label sorted_member_append_ih)

(assume "sorted(cons(c,x))")
(label sorted_member_append_premise1)
(assume "sorted(y)")
(label sorted_member_append_premise2)
(assume "all b.member(b,cons(c,x)) imp leq(b,a)")
(label sorted_member_append_premise3)
(assume "all b.member(b,y) imp not leq(b,a)")
(label sorted_member_append_premise4)

(rw sorted_member_append_premise1 (use listinfo) (open sorted))
(rw "sorted(ku)" (open sorted))
(derive "sorted(x)" (-2 *))

(assume "member(b,x)")
(derive "member(b,cons(c,x))" * (use listinfo) (open member))
(ue ((b . b)) sorted_member_append_premise3 (use *))
```

```
(ci -3 *)
(derive "all b,member(b,x) imp leq(b,a)" *)

(ue ((a . a) (y . y))
    sorted_member_append_ih
    (use -6 sorted_member_append_premise2 * sorted_member_append_premise4)
    (use listinfo))
(label sorted_member_append_lemma)

(assume "x = ku")
(rw sorted_member_append_lemma (use *) (use listinfo) (open append))
(ue ((b . c)) sorted_member_append_premise3
    (use listinfo) (open member))
(trw "sorted(append(cons(c,x),cons(a,y)))"
     (use -3) (use listinfo)
     (open append) (open append)
     (use * -2)
     (open sorted))
(label sorted_member_append_case1)

(assume "not x = ku")
(rw sorted_member_append_lemma (use *) (use listinfo) (open append))
(rw sorted_member_append_premise1 (use -2) (use listinfo) (open sorted))
(trw "sorted(append(cons(c,x),cons(a,y)))"
     (use * -2 -3) (use listinfo)
     (open append) (open append) (open sorted))
(label sorted_member_append_case2)

(derive "x = ku or not x = ku")
(cases * sorted_member_append_case1 sorted_member_append_case2)

(ci (sorted_member_append_premise1
     sorted_member_append_premise2
     sorted_member_append_premise3
     sorted_member_append_premise4)
    *)
(ci sorted_member_append_ih)

(ue ((p . "lambda x,all a y,sorted(x) & sorted(y) &
                            (all b,member(b,x) imp leq(b,a)) &
                            (all b,member(b,y) imp not leq(b,a)) imp
                            sorted(append(x,cons(a,y)))"))
    listind
    (use sorted_member_append_base *)
    (use listinfo))
(label sorted_member_append)
```

```
;;;;    PARTITION_LEMMAS.LSP
;;;;
;;;;            some properties of partition

(proof partition_lemmas)


:Theorem (partl_preserves_shorter):
:all x y.shorter(x,y) imp (all a.shorter(partl(a,x),y))

(trw "shorter(ku,y) imp shorter(partl(a,ku),y)"
     (use listinfo) (open partl) (open shorter))
(label partl_preserves_shorter_base)

(assume "all y.shorter(x,y) imp (all a.shorter(partl(a,x),y))")
(label partl_preserves_shorter_ih)

(assume "shorter(cons(b,x),y)")
(label partl_preserves_shorter_premise)
(rw * (use listinfo) (open shorter))
(label partl_preserves_shorter_lemma)

(assume "leq(b,a)")
(trw "shorter(partl(a,cons(b,x)),y)"
     (use *)
     (use partl_preserves_shorter_lemma)
     (use partl_preserves_shorter_ih)
     (use listinfo) (open partl) (open shorter))
(label partl_preserves_shorter_case1)

(assume "not leq(b,a)")
(ue ((x . "x") (y . "tail(y)") (a . "head(y)")) shorter_lemma1
    (use partl_preserves_shorter_lemma) (use listinfo))
(trw "shorter(partl(a,cons(b,x)),y)"
     (use -2 *)
     (use partl_preserves_shorter_ih)
     (use listinfo) (open partl))
(label partl_preserves_shorter_case2)

(derive "leq(b,a) or not leq(b,a)")
(cases * partl_preserves_shorter_case1 partl_preserves_shorter_case2)

(derive "all a.shorter(partl(a,cons(b,x)),y)" *)
(ci partl_preserves_shorter_premise *)
(derive "all y.shorter(cons(b,x),y) imp
                (all a.shorter(partl(a,cons(b,x)),y))"
        *)
(ci partl_preserves_shorter_ih *)
(ue ((p . "lambda x.(all y.shorter(x,y) imp
                           (all a.shorter(partl(a,x),y)))")
    listind
    (use partl_preserves_shorter_base *) (use listinfo))
(label partl_preserves_shorter)


:Theorem (partr_preserves_shorter):
:all x y.shorter(x,y) imp (all a.shorter(partr(a,x),y))

(trw "shorter(ku,y) imp shorter(partr(a,ku),y)"
     (use listinfo) (open partr) (open shorter))
(label partr_preserves_shorter_base)
```

```
(assume "all y,shorter(x,y) imp (all a,shorter(partr(a,x),y))")
(label partr_preserves_shorter_ih)

(assume "shorter(cons(b,x),y)")
(label partr_preserves_shorter_premise)
(rw * (use listinfo) (open shorter))
(label partr_preserves_shorter_lemma)

(assume "not leq(b,a)")
(trw "shorter(partr(a,cons(b,x)),y)"
     (use *)
     (use partr_preserves_shorter_lemma)
     (use partr_preserves_shorter_ih)
     (use listinfo) (open partr) (open shorter))
(label partr_preserves_shorter_case1)

(assume "leq(b,a)")
(ue ((x . "x") (y . "tail(y)") (a . "head(y)")) shorter_lemma1
    (use partr_preserves_shorter_lemma) (use listinfo))
(trw "shorter(partr(a,cons(b,x)),y)"
     (use -2 *)
     (use partr_preserves_shorter_ih)
     (use listinfo) (open partr))
(label partr_preserves_shorter_case2)

(derive "not leq(b,a) or leq(b,a)")
(cases * partr_preserves_shorter_case1 partr_preserves_shorter_case2)

(derive "all a,shorter(partr(a,cons(b,x)),y)" *)
(ci partr_preserves_shorter_premise *)
(derive "all y,shorter(cons(b,x),y) imp
               (all a,shorter(partr(a,cons(b,x)),y))"
        *)
(ci partr_preserves_shorter_ih *)
(ue ((p . "lambda x,(all y,shorter(x,y) imp
                           (all a,shorter(partr(a,x),y)))")))
    listind
    (use partr_preserves_shorter_base *) (use listinfo))
(label partr_preserves_shorter)


;Theorem (partl_leq_key):
;all a x b,member(b,partl(a,x)) imp leq(b,a)

(trw "member(b,partl(a,ku))" (use listinfo) (open partl) (open member))
(label partl_leq_key_base)

(assume "all a b,member(b,partl(a,x)) imp leq(b,a)")
(label partl_leq_key_ih)

(assume "member(b,partl(a,cons(c,x)))")
(label partl_leq_key_premise)

(assume "leq(c,a)")

(rw partl_leq_key_premise
    (use *) (use listinfo) (open partl) (open member))
(derive "leq(b,a)" (-2 -1 partl_leq_key_ih))
(label partl_leq_key_case1)
```

```
(assume "not leq(c,a)")
(rw partl_leq_key_premise
    (use *) (use listinfo) (open partl))
(derive "leq(b,a)" (* partl_leq_key_ih) (use listinfo))
(label partl_leq_key_case2)

(derive "leq(c,a) or not leq(c,a)")
(cases * partl_leq_key_case1 partl_leq_key_case2)

(ci partl_leq_key_premise *)
(derive "all a b.member(b,partl(a,cons(c,x))) imp leq(b,a)" *)
(ci partl_leq_key_ih *)

(ue ((p . "lambda x.all a b.member(b,partl(a,x)) imp leq(b,a)")) listind
    (use *) (use partl_leq_key_base) (use listinfo))
(label partl_leq_key)


;Theorem (partr_not_leq_key):
;all a x b.member(b,partr(a,x)) imp not leq(b,a)

(trw "member(b,partr(a,ku))" (use listinfo) (open partr) (open member))
(label partr_not_leq_key_base)

(assume "all a b.member(b,partr(a,x)) imp not leq(b,a)")
(label partr_not_leq_key_ih)

(assume "member(b,partr(a,cons(c,x)))")
(label partr_not_leq_key_premise)

(assume "not leq(c,a)")

(rw partr_not_leq_key_premise
    (use *) (use listinfo) (open partr) (open member))
(derive "not leq(b,a)" (-2 -1 partr_not_leq_key_ih))
(label partr_not_leq_key_case1)

(assume "leq(c,a)")
(rw partr_not_leq_key_premise
    (use *) (use listinfo) (open partr))
(derive "not leq(b,a)" (* partr_not_leq_key_ih) (use listinfo))
(label partr_not_leq_key_case2)

(derive "not leq(c,a) or leq(c,a)")
(cases * partr_not_leq_key_case1 partr_not_leq_key_case2)

(ci partr_not_leq_key_premise *)
(derive "all a b.member(b,partr(a,cons(c,x))) imp not leq(b,a)" *)
(ci partr_not_leq_key_ih *)

(ue ((p . "lambda x.all a b.member(b,partr(a,x)) imp not leq(b,a)")) listind
    (use *) (use partr_not_leq_key_base) (use listinfo))
(label partr_not_leq_key)


;Theorem (shorter_partl_head_tail):
;all x. x != ku & tail(x) != ku imp shorter(partl(head(x),tail(x)),x)

(assume "x != ku & tail(x) != ku")
(label shorter_partl_head_tail_assumption)
```

```
(ue ((a . "head(x)") (x . "tail(x)")) shorter_lemma2
    (use listinfo) (use *))

(ue ((x . "tail(x)") (y . "x")) partl_preserves_shorter
    (use *)
    (use shorter_partl_head_tail_assumption)
    (use listinfo))
(ue ((a . "head(x)")) *
    (use shorter_partl_head_tail_assumption)
    (use listinfo))
(ci shorter_partl_head_tail_assumption *)

(derive "all x. x != ku & tail(x) != ku imp shorter(partl(head(x),tail(x)),x)"
        *)
(label shorter_partl_head_tail)


;Theorem (shorter_partr_head_tail):
;all x. x != ku & tail(x) != ku imp shorter(partr(head(x),tail(x)),x)

(assume "x != ku & tail(x) != ku")
(label shorter_partr_head_tail_assumption)

(ue ((a . "head(x)") (x . "tail(x)")) shorter_lemma2
    (use listinfo) (use *))

(ue ((x . "tail(x)") (y . "x")) partr_preserves_shorter
    (use *)
    (use shorter_partr_head_tail_assumption)
    (use listinfo))
(ue ((a . "head(x)")) *
    (use shorter_partr_head_tail_assumption)
    (use listinfo))
(ci shorter_partr_head_tail_assumption *)

(derive "all x. x != ku & tail(x) != ku imp shorter(partr(head(x),tail(x)),x)"
        *)
(label shorter_partr_head_tail)


;Theorem (member_partl_partr_if):
;all x a b.member(b,x) imp
;          member(b,partl(a,x)) or member(b,partr(a,x))

(trw !member(b,ku)! (use listinfo) (open member))

(ue ((p . "lambda x.all a b.member(b,x) imp
                            member(b,partl(a,x)) or member(b,partr(a,x))"))
    listind
    (use *) (use listinfo))
(label member_partl_partr_if_step)

(assume !all a b.member(b,x) imp
                member(b,partl(a,x)) or member(b,partr(a,x))!)
(label member_partl_partr_if_ih)
(ue ((a . a) (b . b)) *)
(label member_partl_partr_if_lemma)

(assume "member(b,cons(c,x))")
(label member_partl_partr_if_premise)
```

```
(rw * (use listinfo) (open member))
(label member_part1_partr_if_premise_lemma)

(assume !leq(c,a)!)

(trw !member(b,part1(a,cons(c,x))) or member(b,partr(a,cons(c,x)))!
     (use *) (use listinfo) (open part1) (open partr))
(rw * (part 2 (part 1 (use listinfo) (open member))))
(rw * (part 2 (der member_part1_partr_if_premise_lemma
                   member_part1_partr_if_lemma)))
(label member_part1_partr_if_case1)

(assume !not leq(c,a)!)

(trw !member(b,part1(a,cons(c,x))) or member(b,partr(a,cons(c,x)))!
     (use *) (use listinfo) (open part1) (open partr))
(rw * (part 2 (part 2 (use listinfo) (open member))))
(rw * (part 2 (der member_part1_partr_if_premise_lemma
                   member_part1_partr_if_lemma)))
(label member_part1_partr_if_case2)

(derive "leq(c,a) or not leq(c,a)")
(cases * member_part1_partr_if_case1 member_part1_partr_if_case2)

(ci member_part1_partr_if_premise *)
(derive
 "all a b.member(b,cons(c,x)) imp
          member(b,part1(a,cons(c,x))) or member(b,partr(a,cons(c,x)))"
 *)
(ci member_part1_partr_if_ih *)
(rw member_part1_partr_if_step (use *))
(label member_part1_partr_if)


;Theorem (member_part1_partr_only_if):
;all x a b.member(b,part1(a,x)) or member(b,partr(a,x)) imp
;          member(b,x)

(trw !member(b,part1(a, ku))! (use listinfo) (open part1) (open member))
(trw !member(b,partr(a, ku))! (use listinfo) (open partr) (open member))

(ue ((p .  "lambda x.all a b.member(b,part1(a,x)) or member(b,partr(a,x)) imp
                             member(b,x)"))
    listind
    (use -2 -1) (use listinfo))
(label member_part1_partr_only_if_step)

(assume !all a b.member(b,part1(a,x)) or member(b,partr(a,x)) imp
                member(b,x)!)
(label member_part1_partr_only_if_ih)
(ue ((a . a) (b . b)) *)
(label member_part1_partr_only_if_lemma)

(assume "member(b,part1(a,cons(c,x))) or member(b,partr(a,cons(c,x)))")
(label member_part1_partr_only_if_premise)

(assume !leq(c,a)!)
(rw member_part1_partr_only_if_premise
    (use *) (use listinfo)
    (part 1 (open part1) (open member))
    (part 2 (open partr)))
```

```
(derive "c = b or member(b,x)"
        (* member_part1_partr_only_if_lemma)
        (use listinfo))
(trw "member(b,cons(c,x))" (use *) (use listinfo) (open member))
(label member_part1_partr_only_if_case1)

(assume !not leq(c,a)!)
(rw member_part1_partr_only_if_premise
    (use *) (use listinfo)
    (part 1 (open part1))
    (part 2 (open partr) (open member)))
(derive "c = b or member(b,x)"
        (* member_part1_partr_only_if_lemma)
        (use listinfo))
(trw "member(b,cons(c,x))" (use *) (use listinfo) (open member))
(label member_part1_partr_only_if_case2)

(derive "leq(c,a) or not leq(c,a)")
(cases * member_part1_partr_only_if_case1 member_part1_partr_only_if_case2)

(ci member_part1_partr_only_if_premise *)
(derive
 "all a b,member(b,part1(a,cons(c,x))) or member(b,partr(a,cons(c,x))) imp
          member(b,cons(c,x))"
 *)
(ci member_part1_partr_only_if_ih *)
(rw member_part1_partr_only_if_step (use *))
(label member_part1_partr_only_if)
```

```
;;;;    MACROS.LSP


(defmacro use_ih (ih assumption label)
  `(progn
    (ue ((x . x)) shorter_partl_head_tail
        (use ,assumption)
        (use listinfo))
    (ue ((y . "partl(head(x),tail(x))")) ,ih
        (use *) (use listinfo) (use ,assumption))
    (label ,label)
    (ue ((x . x)) shorter_partr_head_tail
        (use ,assumption)
        (use listinfo))
    (ue ((y . "partr(head(x),tail(x))")) ,ih
        (use *) (use listinfo) (use ,assumption))
    (label ,label)))

(defmacro use_ihl (ih assumption label)
  `(progn
    (ue ((x . x)) shorter_partl_head_tail
        (use ,assumption)
        (use listinfo))
    (ue ((y . "partl(head(x),tail(x))") (a . a)) ,ih
        (use *) (use listinfo) (use ,assumption))
    (label ,label)
    (ue ((x . x)) shorter_partr_head_tail
        (use ,assumption)
        (use listinfo))
    (ue ((y . "partr(head(x),tail(x))") (a . a)) ,ih
        (use *) (use listinfo) (use ,assumption))
    (label ,label)))

(defmacro cases3 (case1 case2 case3)
  `(progn
    (derive "x = ku or not x = ku & tail(x) = ku or
             not( x = ku & not tail(x) = ku"
            listinfo)
    (cases * ,case1 ,case2 ,case3)))
```

```
;;;;    QSORT_IS_TOTAL.LSP

(proof qsort_is_total)

;Theorem (qsort_is_total):
;all x.list(qsort(x))

(axiom "all x, x != ku & tail(x) != ku imp shorter(partl(head(x),tail(x)),x)")
(label shorter_partl_head_tail)
;This theorem is proved in PARTITION_LEMMAS.LSP.

(axiom "all x, x != ku & tail(x) != ku imp shorter(partr(head(x),tail(x)),x)")
(label shorter_partr_head_tail)
;This theorem is proved in PARTITION_LEMMAS.LSP.


(assume "all y.shorter(y,x) imp list(qsort(y))")
(label qsort_is_total_ih)

(assume "x = ku")
(trw "list(qsort(x))" (use listinfo) (open qsort) (use *))
(label qsort_is_total_case1)

(assume "not x = ku & tail(x) = ku")
(trw "list(qsort(x))" (use listinfo) (open qsort) (use *))
(label qsort_is_total_case2)

(assume "not x = ku & not tail(x) = ku")
(label qsort_is_total_case3_assumption)

(use_ih qsort_is_total_ih
        qsort_is_total_case3_assumption
        qsort_is_total_case3_partition)

(trw "list(qsort(x))"
     (use listinfo)
     (open qsort)
     (use qsort_is_total_case3_assumption)
     (use qsort_is_total_case3_partition))
(label qsort_is_total_case3)

(cases3 qsort_is_total_case1 qsort_is_total_case2 qsort_is_total_case3)
(ci qsort_is_total_ih *)
(ue ((p . "lambda x.list(qsort(x))")) shorterind (use *))
(label qsort_is_total)
(label listinfo)
```

```
;;;;    QSORT_SORTS.LSP

(proof qsort_sorts)


(axiom "all x.list(qsort(x))")
(label listinfo)
;This theorem is proved in QSORT_IS_TOTAL.LSP.

(axiom "all a x b.member(b,part1(a,x)) imp leq(b,a)")
(label part1_leq_key)
;This theorem is proved in PARTITION_LEMMAS.LSP.

(axiom "all a x b.member(b,partr(a,x)) imp not leq(b,a)")
(label partr_not_leq_key)
;This theorem is proved in PARTITION_LEMMAS.LSP.

(axiom "all x.x != ku & tail(x) != ku imp shorter(part1(head(x),tail(x)),x)")
(label shorter_part1_head_tail)
;This theorem is proved in PARTITION_LEMMAS.LSP.

(axiom "all x. x != ku & tail(x) != ku imp shorter(partr(head(x),tail(x)),x)")
(label shorter_partr_head_tail)
;This theorem is proved in PARTITION_LEMMAS.LSP.

(axiom "all a x b.member(b,x) imp
                  member(b,part1(a,x)) or member(b,partr(a,x))")
(label member_part1_partr_if)
;This theorem is proved in PARTITION_LEMMAS.LSP.

(axiom "all a x b.member(b,part1(a,x)) or member(b,partr(a,x)) imp
                  member(b,x)")
(label member_part1_partr_only_if)
;This theorem is proved in PARTITION_LEMMAS.LSP.


;Theorem (member_qsort_if):
;all x a.member(a,qsort(x)) imp member(a,x)

(assume "all y.shorter(y,x) imp (all a.member(a,qsort(y)) imp member(a,y))")
(label member_qsort_if_ih)

(assume "x = ku")
(trw "member(a,qsort(x)) imp member(a,x)"
     (part 1 (use *) (use listinfo) (open qsort) (open member)))
(label member_qsort_if_case1)

(assume "not x = ku & tail(x) = ku")
(trw "member(a,qsort(x)) imp member(a,x)"
     (use *) (use listinfo) (open qsort) (open member))
(label member_qsort_if_case2)

(assume "not x = ku & not tail(x) = ku")
(label member_qsort_if_case3_assumption)

(use_ihl member_qsort_if_ih
         member_qsort_if_case3_assumption
         member_qsort_if_case3_partition)

(assume "member(a,qsort(x))")
(label member_qsort_if_case3_premise)
```

```
(rw * (use member_qsort_if_case3_assumption)
      (use listinfo) (open qsort))


(ue ((a . "head(x)")
     (x . "qsort(partl(head(x),tail(x)))")
     (y . "qsort(partr(head(x),tail(x)))")
     (b . a))
    member_append_lemma_if
    (use *) (use member_qsort_if_case3_assumption))
(rw * (use member_qsort_if_case3_assumption) (use listinfo))
(derive "member(a,partl(head(x),tail(x))) or a = head(x) or
         member(a,partr(head(x),tail(x)))"
        (* member_qsort_if_case3_partition))

(ue ((a . "head(x)") (x . "tail(x)") (b . a))
    member_partl_partr_only_if
    (use member_qsort_if_case3_assumption)
    (use listinfo))
(derive "member(a,tail(x)) or a = head(x)" (-2 *))

(derive "member(a,x)" *
        (use member_qsort_if_case3_assumption)
        (use listinfo)
        (open member))
(ci member_qsort_if_case3_premise *)
(label member_qsort_if_case3)

(cases3 member_qsort_if_case1 member_qsort_if_case2 member_qsort_if_case3)
(ci member_qsort_if_ih *)

(ue ((p . "lambda x,all a,member(a,qsort(x)) imp member(a,x)"))
    shorterind
    (use *)
    (use listinfo))
(label member_qsort_if)


:Theorem (member_qsort_only_if):
;all x a,member(a,x) imp member(a,qsort(x))

(assume "all y,shorter(y,x) imp (all a,member(a,y) imp member(a,qsort(y)))")
(label member_qsort_only_if_ih)

(assume "x = ku")
(trw "member(a,x) imp member(a,qsort(x))"
     (part 1 (use *) (use listinfo) (open qsort) (open member)))
(label member_qsort_only_if_case1)

(assume "not x = ku & tail(x) = ku")
(trw "member(a,x) imp member(a,qsort(x))"
     (use *) (use listinfo) (open qsort) (open member))
(label member_qsort_only_if_case2)

(assume "not x = ku & not tail(x) = ku")
(label member_qsort_only_if_case3_assumption)

(use_ihl member_qsort_only_if_ih
         member_qsort_only_if_case3_assumption
         member_qsort_only_if_case3_partition)
```

```
(assume "member(a,x)")
(label member_qsort_only_if_case3_premise)
(rw * (use member_qsort_only_if_case3_assumption)
      (use listinfo) (open member))

(ue ((a . "head(x)") (x . "tail(x)") (b . a))
    member_part1_partr_if
    (use member_qsort_only_if_case3_assumption)
    (use listinfo))

(derive "member(a,part1(head(x),tail(x))) or a = head(x) or
         member(a,partr(head(x),tail(x)))"
        (-2 -1))

(derive "member(a,qsort(part1(head(x),tail(x)))) or a = head(x) or
         member(a,qsort(partr(head(x),tail(x))))"
        (* member_qsort_only_if_case3_partition))

(ue ((a . "head(x)")
     (x . "qsort(part1(head(x),tail(x)))")
     (y . "qsort(partr(head(x),tail(x)))")
     (b . a))
    member_append_lemma_only_if
    (use *) (use member_qsort_only_if_case3_assumption) (use listinfo))
(trw "member(a,qsort(x))"
     (use *) (open qsort)
     (use member_qsort_only_if_case3_assumption)
     (use listinfo))

(ci member_qsort_only_if_case3_premise *)
(label member_qsort_only_if_case3)

(cases3 member_qsort_only_if_case1
        member_qsort_only_if_case2
        member_qsort_only_if_case3)
(ci member_qsort_only_if_ih *)

(ue ((p . "lambda x.all a.member(a,x) imp member(a,qsort(x))"))
    shorterind
    (use *)
    (use listinfo))
(label member_qsort_only_if)


;Theorem (sorted_qsort):
;all x.sorted(qsort(x))

(assume "all y.shorter(y,x) imp sorted(qsort(y))")
(label sorted_qsort_ih)

(assume "x = ku")
(trw "sorted(qsort(x))"
     (use *) (use listinfo) (open qsort) (open sorted))
(label sorted_qsort_case1)

(assume "not x = ku & tail(x) = ku")
(trw "sorted(qsort(x))"
     (use *) (use listinfo) (open qsort) (open sorted))
(label sorted_qsort_case2)

(assume "not x = ku & not tail(x) = ku")
```

```
(label sorted_qsort_case3_assumption)

(use_ih sorted_qsort_ih
        sorted_qsort_case3_assumption
        sorted_qsort_case3_partition)

(assume "member(b,qsort(partl(head(x),tail(x))))")
(ue ((x . "partl(head(x),tail(x))") (a . b)) member_qsort_if
    (use *) (use sorted_qsort_case3_assumption) (use listinfo))
(ue ((a . "head(x)") (x . "tail(x)") (b . b)) partl_leq_key
    (use *) (use sorted_qsort_case3_assumption) (use listinfo))
(ci -3 *)

(assume "member(b,qsort(partr(head(x),tail(x))))")
(ue ((x . "partr(head(x),tail(x))") (a . b)) member_qsort_if
    (use *) (use sorted_qsort_case3_assumption) (use listinfo))
(ue ((a . "head(x)") (x . "tail(x)") (b . b)) partr_not_leq_key
    (use *) (use sorted_qsort_case3_assumption) (use listinfo))
(ci -3 *)

(ue ((a . "head(x)")
     (x . "qsort(partl(head(x),tail(x)))")
     (y . "qsort(partr(head(x),tail(x)))"))
    sorted_member_append
    (use sorted_qsort_case3_partition)
    (use -5 -1)
    (use sorted_qsort_case3_assumption)
    (use listinfo))

(trw "sorted(qsort(x))"
     (use *)
     (use sorted_qsort_case3_assumption)
     (use listinfo)
     (open qsort))
(label sorted_qsort_case3)

(cases3 sorted_qsort_case1 sorted_qsort_case2 sorted_qsort_case3)

(ci sorted_qsort_ih *)

(ue ((p . "lambda x.sorted(qsort(x))"))
    shorterind
    (use *)
    (use listinfo))
(label sorted_qsort)
```

```
(wipe-out)
(proof real)

(decl plus (type !(ground,ground,ground*):ground!)
      (syntype constant) (infixname +)
      (bindingpower 930) (associativity both))

(decl times (type !(ground,ground,ground*):ground!)
      (syntype constant) (infixname *)
      (bindingpower 935) (associativity both))

(decl (zero unit) (syntype constant) (type !ground!))

(decl (aa bb cc)  (type !ground!))

(axiom !all aa,( aa + zero = aa &  zero + aa = aa)!)
(label simpinfo)

(axiom !all aa,( aa * unit = aa &  unit * aa = aa)!)
(label simpinfo)

(decl (ff gg hh) (type !ground:ground!))

;simpfacts

(axiom !zero != unit!)
(label simpinfo)

;multiplication

(axiom !all aa. aa * zero = zero & zero * aa = zero!)
(label simpinfo)

(axiom !all aa. aa * unit = aa & unit * aa = aa!)
(label simpinfo)

(axiom !all aa bb. aa * bb = bb * aa!)
(label product_commute)(label commute)


;addition

(axiom !all aa. aa + zero = aa!)
(label simpinfo)

(axiom !all aa bb. aa + bb = bb + aa!)
(label plus_commute)(label commute)

(axiom !all aa bb cc.aa + bb = aa + cc iff bb = cc!)
(label simpinfo)(label plus_cancel)

(axiom !all aa bb cc.bb + aa = cc + aa iff bb = cc!)
(label simpinfo)(label plus_cancel)


;extensionality

(axiom !all f g.(all x.f(x) = g(x)) iff f = g!)
(label ext)
```

```
;distributivity
(axiom !all aa bb cc. aa * (bb + cc) = aa * bb + aa * cc!)
(label simpinfo)(label distfacts)

(axiom !all aa bb cc. (aa + bb) * cc = aa * cc + bb * cc!)
(label simpinfo)(label distfacts)

(decl nilpotent (type !ground:truthval!))

(decl (d d1 d2 d3) (type !ground!) (sort !nilpotent!))

(decl derivation
 (type !(ground:ground):(ground:ground)!) (postfixname !'!)
 (bindingpower 990))

(axiom !all f aa d. f(aa + d) = f(aa) + d * (f')(aa)!)
(label taylor)

(decl func_plus (type !((ground:ground),(ground:ground)):(ground:ground)!)
      (infixname ++) (bindingpower 930))

(define func_plus "all f1 f2. f1 ++ f2 = lambda aa. f1(aa) + f2(aa)")
(label func_plusdef)

(decl func_product (type !((ground:ground),(ground:ground)):(ground:ground)!)
      (infixname **) (bindingpower 935))

(define func_product "all f1 f2. f1 ** f2 = lambda aa. f1(aa) * f2(aa)")
(label func_productdef)

(axiom "all d. d * d = zero")
(label simpinfo)

(axiom !all aa bb.(all d.d * aa = d * bb) iff aa = bb!)
(label simpinfo)(label taylor_unique)

(axiom !all aa bb.(all d.aa * d = bb * d) iff aa = bb!)
(label simpinfo)(label taylor_unique)



;;;;;;(f ** g)' = g ** f' ++ f ** g'

(trw !all d.(f ** g)(x+d) = f(x+d)*g(x+d)! (open func_product))
;all d.(f ** g)(x+d) = f(x+d)*g(x+d)

(rw * (use taylor mode exact) (use distfacts mode always) (use product_commute))
;all d.(f ** g)(x)+d*((f ** g)')(x) =
;       f(x)*g(x)+d*g(x)*(f')(x)+d*f(x)*(g')(x)

(rw * (part 1 (part 1 (part 1 (open func_product)))))
;all d.d*((f ** g)')(x) = d*g(x)*(f')(x)+d*f(x)*(g')(x)

(trw !all d.d*g(x)*(f')(x)+d*f(x)*(g')(x) =  d*((g ** f' ++ f ** g')(x))!
     (open func_product) (open func_plus) (use distfacts))
;all d.d*g(x)*(f')(x)+d*f(x)*(g')(x) = d*(g ** f' ++ f ** g')(x)

(rw "-2" (use * mode exact))
;((f ** g)')(x) = (g ** f' ++ f ** g')(x)
```

```
(derive !all x.((f ** g)')(x) = (g ** f' ++ f ** g')(x)! *)

(rw * (use ext))
!(f ** g)' = g ** f' ++ f ** g'


!!!!! (f ++ g)' = f' ++ g'

(trw !all d.(f ++ g)(x+d) = f(x+d)+g(x+d)! (open func_plus))
!all d.(f ++ g)(x+d) = f(x+d)+g(x+d)

(rw * (use taylor mode exact) (use plus_commute))
!all d.(f ++ g)(x)+d*((f ++ g)')(x) = f(x)+g(x)+d*(f')(x)+d*(g')(x)

(rw * (part 1 (part 1 (part 1 (open func_plus)))))
!all d.d*((f ++ g)')(x) = d*(f')(x)+d*(g')(x)
(label tmp1)

(trw !all x d.d*(f')(x)+d*(g')(x) = d*(f' ++ g')(x)!
      ((open func_plus) (use distfacts mode exact)))
!all x d.d*(f')(x)+d*(g')(x) = d*(f' ++ g')(x)
(label tmp2)

(rw tmp1 (use tmp2))
!((f ++ g)')(x) = (f' ++ g')(x)

(derive !all x.((f ++ g)')(x) = (f' ++ g')(x)! *)

(rw * (use ext))
!(f ++ g)' = f' ++ g'
```

```
(wipe-out)
(proof robinson)

(decl st (type !ground:ground!) (syntype constant))

(decl standard (type !ground:truthval!) (syntype constant))

(decl near (type !(ground,ground):truthval!) (syntype constant)
           (infixname !~!) (bindingpower 900))

(decl compact (type !(ground:truthval):truthval!) (syntype constant))

(define compact "all p. (compact(p) = all x.(p(x) imp p(st(x))))")
(label compactdef)

(decl cont (type !((ground:ground),ground):truthval!) (syntype constant))

(define cont
  "all f r.(cont(f,r) = all y.(((standard(r) & r ~ y) imp f(r) ~ f(y))))")

(decl unf_cont (type !((ground:ground),(ground:truthval)):truthval!)
  (syntype constant))

(define unf_cont
 "all f p.(unf_cont(f,p) = all x y.(((p(x) & p(y) & x ~ y) imp f(x) ~ f(y))))")(label

(axiom "all x. x ~ x")
(label near_eq)

(axiom "all x y. x ~ y iff y ~ x")
(label near_eq)

(axiom "all x y z. (x ~ y & y ~ z) imp x ~ z")
(label near_eq)(label near_trans)

(axiom "all x. standard(st(x))")
(label simpinfo)

(axiom "all x.st(x)~x")
(label simpinfo)


;;; PROOF

(assume !p(x)&p(y)!)
;deps: (asp1)
(label asp1)

(assume !all x.p(x) imp p(st(x))!)
;deps: (asp2)
(label asp2)

(assume !x~y!)
(label asp3 )
;deps: (asp3)

(trw !st(x)~x&x~y imp st(x)~y! ((nuse simpinfo) (use near_trans)))
;st(x)~x&x~y imp st(x)~y

(rw * (use asp3))
;st(x)~y
```

```
;deps: (asp3)

(ue ((x . !x!)) asp2 (use asp1))
;p(st(x))
;deps: (asp1 asp2)

;labels: asp4
(assume !all x.p(x) imp cont(f,x)!)
;deps: (asp4)
(label asp4)

(rw asp4 (open cont))
;all x.p(x) imp (all y.standard(x)&x~y imp f(x)~f(y))
;deps: (asp4)

(ue ((x . !st(x)!)) * nil)
;p(st(x)) imp (all y.st(x)~y imp f(st(x))~f(y))
;deps: (asp4)

(rw * (use "-4"))
;all y.st(x)~y imp f(st(x))~f(y)
;deps: (asp1 asp2 asp4)
(label hlp11)

(ue ((y . !y!)) * nil)
;st(x)~y imp f(st(x))~f(y)
;deps: (asp1 asp2 asp4)

(rw * (use "-7"))
;f(st(x))~f(y)
;deps: (asp1 asp2 asp3 asp4)

(ue ((y . x)) hlp11)

(derive "f(x)~f(y)" (11 12 * "-2"))  28

(ci (asp1 asp3) * nil)
;p(x)&p(y)&x~y imp f(x)~f(y)
;deps: (asp2 asp4)

(derive !all x y.p(x)&p(y)&x~y imp f(x)~f(y)! *)
;deps: (asp2 asp4)

(ci (asp2 asp4) *)
;(all x.p(x) imp p(st(x)))&(all x.p(x) imp cont(f,x)) imp
;(all x y.p(x)&p(y)&x~y imp f(x)~f(y))

(rw * ((use unf_contdef direction reverse)
       (use compactdef direction reverse)))
;compact(p)&(all x.p(x) imp cont(f,x)) imp unf_cont(f,p)

;;;QED
```