December 1986

Report No. STAN-CS-87-1155

Experiments in Automatic Theorem Proving

by

Gianluigi Bellin

Jussi Ketonen

Department of Computer Science

Stanford University
Stanford, CA 94305

Experiments in Automatic Theorem Proving

by

Gianluigi Bellin and Jussi Ketonen

Research Sponsored by

Advanced Research Projects Agency
and
National Science Foundation

Department of Computer Science
Stanford University
Stanford, CA 94305

This version of the PERM was printed on December 20, 1986.

CONTENTS

CONTENTS

INTRODUCTION

1. Introduction.

1.1. Motivation and general comments.

The best guarantee is to find programs which are not too hard to execute, but can be applied to very "many" instances of interest. G.Kreisel [1981]

The question of finding convenient representations for mathematical facts is one of the most interesting challenges in the field of mechanical theorem proving. A solution should lead naturally to other applications. Thus, given a problem? instead of simply asking whether one can find a particular representation that enables a machine to solve it, one should also ask:

— Are we learning something else from this experiment,. besides the fact that the (usually well known) theorem is true?

— Is our representation abstract enough to allow applications of the result to similar problems'? For otherwise, given the well known present limitations of mechanical theorem provers, it is hard to imagine that the natural customer of the technology of automatic proof-checking. i.e. the working mathematician or teacher of mathematics, may ever find any appeal in it.

In the esperiments described in this paper we have tried to meet this challenge by using the proof checker EKL. a. system whose flexibility is increased by the use of high order logic. Using the exprèssive power of EKL we abstractly represent a result in second order language, prove it and then apply it in a. natural way to different contests.

The focus of our experiment is the basic theory of permutations. A permutation is a bijection of a (finite) set into itself. Our aim is to prove that permutations of a finite set, with the operation of composition of functions, form a group. Specifically, given a. finite set S, we want to show that

(I) the composition f o g of two permutations f and g on S is a. permutation on S and composition is associative;

(2) the identity function i on S is a permutation; for any permutation f on S, i o $f = f$ (*left identity*) and f o $i = f$ (*right identity*)

(3) if f is a permutation on S, then there is a permutation f^{-1} of S such that f^{-1} 0 $f = i$ (*left* inverse) and f o $f^{-1} = i$ (*right inverse*).

EKL can easily express such facts in first or higher order logic. We can simply prove the facts stated above using elementary set theory. In the proof we need the 'pigeon hole' principle of elementary arithmetic: if we want to fill each of n holes and we have only n objects, then no hole can contain more than one object. The proof of this fact is not entirely trivial. Although it can be formulated in the language of first order logic with symbols for order, successor and a function symbol, it cannot be proved in the fragment of arithmetic having the usual axioms for order and successor plus induction applied to unary formulas only [Goad 1979]. Our proof of the pigeon hole principle, expressed in second order arithmetic, presents no such difficulties, since we do not restrict the inductive principle available to EKL.

The mathematical notions considered here do not require higher order logic in an 'essential way'. Any fact stated herein could be rephrased in terms of first order logic. Rather, the expressive power of EKL is used to emphasize the freedom in the choice of representations and flexibility proving facts.

There are many ways of representing finite functions. Having chosen one, we must define the operations of composition of functions, the identity function and the operation of taking the inverse of a function. Then we must prove the facts (1) - (3) using that interpretation.

We can always assume that a representation of a finite set of objects gives also an enumeration of it. Therefore we may represent finite functions as *lists* and use the axioms of LISP to prove facts about them.

We give two different representations of permutations, one using association lists and the other using lists of numbers. In the first case the association list contains the graph of the function. Domain and range are represented by lists obtained in the obvious way from the given association list. In the second case the list contains the range in the order given by the domain. The domain is not represented by a list: rather it is a segment. of the set of natural numbers. In this sense we have a more abstract. representation, in which it is slightly easier to apply the pigeon hole principle as an abstract fact of arithmetic. This representation has been traditionally used in mathematics in order to tall; about. finite permutations.

The application of the pigeon hole principle occurs at similar points of the proofs, but the second order statement expressing it is instantiated by different functions. The improvement of efficiency obtained by higher order logic is particularly obvious here.

We also give two versions of the results for representations using lists of numbers. In the first version the operations of composition, identity and inverse are defined by predicates: we shall call it Permutation-Predicate or PERMP. In the second these operations are defined by functions: we shall call this approach PERMF. for Permutation-Function.

The contrast between the representations through predicates and through functions is an aspect of the tension between extensional and intensional approaches to mathematics. This is relevant in general to the automatic verification of the correctness of programs. The way we dealt. with this tension can be taken. in some sense. as the 'moral' of our esperiment. We try to summarize oui point iii the folloning (idealized) history of the project.

Suppose we have written a. LISP program for permutations. using any representation and we want to prove it correct 'by pencil and paper'. If we are willing to assume the pigeon hole principle as evident and to justify the inferences by the label 'evident by elementary arithmetic', then the 'proof of correctness is fairly simple, no matter what representation one chooses. Only the forms of the intluctions require some thought.

On the other hand, if we try to check our proof mechanically, say using EKL, and have in our proof library only simple facts of arithmetic and of LISP, then the task may look discouraging. Too many facts of elementary arithmetic and LISP functions may be needed, especially if we stick to the original form of our recursive programs in a 'too constructive' fashion.

This feeling of uneasiness is well known and perhaps unavoidable in the early stage of such enterprise as ours: since the first efforts of large scale formalization of elementary mathematics (e.g. Russell and Whitehead's "Principia"), it became obvious that the amount. of innocent presuppositions hidden in intuitive arguments grows to the size of tropical forest iii a full formalization. However, our experiments and many others show that some nontrivial results are indeed provable. when the basic proof libraries are reasonably furnished. It is also likely that. simple improvements of EKL -more semantic attachments—will make our task easier.

Minor details in the choice of the representation and in the formulation of the results may have major consequences in terms of length of the proofs and feasibility of the project. For instance, the representation of permutations in terms of association lists makes most proofs easy applications of one induction principle. induction on association lists. However, more work is needed to show that these facts on association lists actually establish tlie desired facts about permutations: indeed the representation by association list--unlike that by lists of numbers—is not unique. A permutation is represented by an equivalence class of association lists. not by a single association list. Hence one needs a canonical way to choose representatives. a *normal form*. that can be obtained e.g. by ordering the field of the permutations. It is reasonable to consider other representations having the uniqueness property.

At first sight there seems to be no question that it. is better to represent our operations by functions rather than by predicates. One can test this assumption by comparing our two versions PERMP and PERMF: to find a confirmation. one just looks at the treatment of composition of permutations and the proof that composition is associative. The operation on lists that represents composition of functions is better represented as a. binary function. defined by recursion on the first list. rather than a ternary predicate. Indeed in the first case we can use a straightforward proof by induction on the recursive definition of the functions, whereas in the second case predicates require some relatively complicated substitutions. Fintling these substitutions would require a huge number of random attempts if they were done without human direction.

Interestingly enough. many other proofs employing list represent&ion axe easier when the notions in question are formulated using predicates rather than functions. This is true especially of proofs about the identity and the inverse of a permutation. In the version PERMP. such proofs are simply obtained by expanding the assumptions and the definitions. In the version PERMF. the recursive definitions may be quite complicated. and the inductive proofs become quite involved.

This situation is in many ways analogous to problem in various areas of mathematics. In the representation through functions the intensional features of our programs are closely represented. On the contrary. iii the representation through predicates only the extensional properties of our functions are relevant. It is well known that in most mathematical practice only extensional facts are considered. We may say that predicates allow slightly more abstract definitions of the operations than functions. In mathematics often a small progress towards abstraction simplifies the presentation considerably.

If we start. our proof of correctness with the definitions contained in the version PERMF, we may find it convenient to look at the definitions of the operations in PERMP and to prove them first as lemmata. One can then use these facts in different contexts instead of going through longer direct proofs.

Abstracting lemmata and breaking arguments into suitable parts is the basis for mathematical communication: it makes proof 'easy to take and easy to remember'. This remark by Kreisel (a variation on a theme by Wittgenstein) is highly appropriate here. The readability of mechanical proofs depends on such devices even more than the readability of 'pencil and paper' proofs. An automatic proof of correctness of previously written program may be too long and tedious for human consumption. A better organization of the problem. based on more abstract consideration of the facts in question, may significantly increase the readability of such proofs.

Some objections may be raised to our remarks. On one side. one may argue that it is not clear what counts as evidence in favour of our claims: isn't it after all just a question of mathematical *taste*?

On the other side, even granting our claims. one may be *a priori* skeptical about the relevance of our investigation. Haven't we simply verified. through mechanical proof checking of mathematically trivial examples the well known fact that there arc good and bad *styles* of mathematical presentation? Can we expect any interesting theoretical discovery to result from experiments of this kind?

In our experiments we search for methods to effectively use the given technology and for guidelines to improve it. Current practice of informal mathematics and theoretical results from logic do not immediately provide all the relevant information. Proof checking is a. practice of interaction between a. user and a. given technology, in which human capacities, technical possibilities, linguistic features and methods of interaction are all relevant,. For instance, we know from the Normalization Theorem in Proof Theory that direct proofs are generally longer than those using lemmata. It is very well possible that different languages or different theorem provers may suggest different strategies of proof checking. In particular. we cannot. rule out the possibility that language may be created, a technology produced and experiment esibited in which, say, most of our lemmata have convenient direct proofs. Only experience can decide. But given a certain technology. practice does indeed show what directions are convenient and what projects feasible. Strategies and methods of proofs, not only the subjective qualities of the user, are decisive in determining the success of a project.

On the other hand, no matter how plausible the reasons of the skeptic may look. the performance of automatic proof checkers has been remarkably improved since the first experiments. Instruments are available that allow a. 'microscopic analysis of mathematical proofs: a certain amont of experimentation has already been performed. The analysis of what is usually regarded as 'style' of presentation may possibly disclose important features of proofs, that have been overlooked so far. Above all, this work is a necessary preliminary step to start applying *automatic proof transformations (e.g.* extraction of bounds, transformation of non elementary proofs into elementary ones, cut elimination and functional interpretation, etc.) to mathematically significant examples. And there, for a. logician. the real fun begins.

1.2.. **The-Proofs in EKL.**

EKL is a. proof checker and constructor that uses a typed language, a rewriting system, a decision procedure and semantic attachments.

The language of EKL is described in detail in the user's manual [Iietonen and Weening 1984]. For the sake of completeness, we will describe some of the basic facets of this system.

Remark 1. EKL does not distinguish between uppercase and lowercase. As a. convention, in this paper we will use lowecase typeweriter-like font for commands and formulas occurring within a command, and uppercase typeweriter-like font for the formulas returned by EKL. The output of EKL is preceded by semicolon. Thus

```
(trw |p⊃p|)
```

is a command (asking EKL to verify a tautology) and

```
;P is unknown.
;the symbol P declared to have tvpe TRUTHVAL
;P⊃P
```

is the answer by EKL. The first, two lines inform us that a default declaration has been made: the third tells that EKL has verified the tautology.

We thank J. McCarthy for his constant support and encouragement. We owe C. Talcott many ideas and suggestions at various stages of the work. Thanks to R. Casley and J. Weening for their fundamental TEXnical help. This research was supported by grants NSF MCS 82-06565 and ARPA N000-39-82-C-0250

Remark 2. EKL commands use the LISP syntax

(funct argl . . . argn)

where the function (command) funct is applied to the arguments `arg1 . . .` argn. In describing such commands we use the espressions *&optional* and *&rest*.

(funct **argl** . . . *&optional* argj . . . *&rest* `param`)

'*&optional*' indicates that all arguments following it are optional and are given a default value if omitted. '*&rest*' means that '`param`' indicates the the list of all arguments following it, rat her t han a single parameter.

1.2.1. The Language specified by EKL.

A list of linguistic attributes, i.e. a *declaration*, is associated with every atom. The main attributes of a declaration are the type, the *syntype* and the *sort.*

The *type* of an EKL object tells how that object can be applied. For example. an object of type ground → ground can be applied to objects of type ground resulting in an object of type ground. An object of type ground* → ground can be applied to any number of objects of type ground resulting in an object of type ground. Thus objects of this type could be regarded as having variable *arity.* A sentence is an object of type truthval. A unary predicate is au object of type `ground` → truthval. Sets can also be represented as objects of this type.

In declaring the type of a new entity, the operator `@` gives the type of a. (previously defined) object. Thus

```
(decl setseq (type: |@n→@set|))
```

establishes that setseq has the type of a. sequence of sets. i.e. the type of a?. function from objects of the type of natural numbers to objects of the type of sets. Since natural numbers have type ground and sets have type `ground→truthval`, the above declaration is the same as

```
(decl setseq (type: |ground→(ground→truthval)|)).
```

The *syntyye* specifies whether a. linguistic object is a *variable* — so that it can be quantified —. a *constant* — so that it cannot be quantified — or a *bindop*, an operator binding varia bles.

A *sort* in EKL is simply a unary predicate. Every EKL symbol has a. sort. The default is universal — the most general sort, of any type.

Typically we may have a variable n of sort natnum and a variable x of sort universal. Then statements like `∀n.P(n)` are equivalent to `∀x .natnum(x)⊃P(x)`.

In existential generalization, X-a.bstraction etc. EKL checks whether the term in quest ion satisfies sort restrictions. For example, the formula. `∀n . A (n) ⊃∀x . A(x)` is not provable in t he above situation, unless facts like `∀x.natnum(x)` are in use.

The information that a function is defined for a certain argument (or that, a. progra m t erminates) can be given as a fact about sorts. In the following example. to prove that `numseq(m)` is of the sort natnum is to show that the function numseq is defined for m as an argument. Of course EKL has cannot determine this just from the declaration of numseq.

```
(proof sums)
(decl (i j k m n) (sort: natnum))
(decl numseq (type: |@n→@n|))

(derive |∀m.natnum(numseq(m)) I)
; failed to derive
NATNUM(NUMSEQ(N))

(trw |∀m.natnum(numseq(m))|)
;(∀M.NATNUM(NUMSEQ(M)))≡(∀M.NATNUM(NUMSEQ(M)))
```

Some EKL symbols are predeclared: we cannot modify their at tributes. We can introduce linguistic objects using the EKL command DECLARE:

(*I*) (decl <symbol> . <attributes>).

If we introduce a new symbol without declaring it. EKL tries a default declaration and tells us what it is.

A *context* is simply a list of declarations for atoms.

1.2.2. Proofs and Lines in EKL.

A proof in EKL consists of lines. Each line in a. proof is a. result of a. command. There are several different types of lines:

(1) Lines that result from declarations. These have the effect of setting the contest of a line and adjoining the declaration to the current contest.

('2) Lines resulting from other commands.

Exa mples:

(*II*) (assume wff)

The formula **wf f** is assumed true, with the above line introduced as a dependency.

(*III*) (axiom wff)

The formula wf f is assumed as true, with no visible dependencies in t roduced.

(*IV*) (defax symbol wff)

The formula wf f is assumed as true and regarded as the definition of symbol.

(*V*) (define symbol wff &*optional* rewriter)

The formula wff is regarded as the definition of symbol. provided that the truth of 3 symbol.wff follows using the rewriter. The formula wff must cont ain symbol.

(*VI*) (trw term &*optional* rewriter)

The term term is rewritten using standard rewriting, the lines labeled previously as simpinf o and the instructions given by rewriter.

Let term1 be the result of such rewriting. If term is a. term then the formula term = term1 is given as conclusion: if term is a formula. then term ≡ term1 is derived. unless term1 is true, in which case term is derived, or false. in which case ¬term is derived.

(*VII*) (rw &*optional* line rewriter)

The line line is rewrit ten using standard rewriting, the lines labeled previously as simpinf o and the instructions given by rewriter.

(*VIII*) (derive term &*optional* linerange rewriter)

The formula term is derived from the formulas in linerange. using the decision procedure. lines previously labeled as s impinf o. st andard rewriting and rewriting according to the instruct ions given by the rewriter.

(*IX*) (cases line linerange)

The lines in linerange must contain the same formula, say A; line must be a disjunction. This command corresponds to the conclusion of a "proof by cases". Suppose we are able to derive A from A_1 and also from A_2 and . . . and also from A,. Suppose we prove $A_1 \vee A_2 \vee \ldots \vee A_n$. Then we can conclude A "independently of" $A_1 ,\ldots, A_n$.

(*X*) (ci linerange &*optional* line rewriter)

Let the lines in linerange contain the formulas Al,.....A_n. Let the formula in line be B. Then the result of this command is

$$A_1 \wedge \ldots \wedge A_n \supset B,$$

and this formula. will not "depend on" $A_1 ,\ldots, A_n$.

(*XI*) (ue termslst &*optional* linedg rewriter)

This corresponds to the instantiation of a universal statement,. If termslst contains the pair (x t). t is of the same type and sort as x and linedg is of the form ∀x . A(x), then the UE command will yield A(t). rewritten according to rewriter (and the lines previously labeled as simpinfo).

Let us say that the variable x is *explicitly universally quantified* in $\forall x.A(x)$. We define below what it means for x to be *implicitly universally quantified* in a line. The ue command is extended to the case of implicitly quantified variables and also to the case of multiple substitution. with termslst being a list of pairs.

1.2.3. Lines and dependencies.

Each line has associated to it its contest and dependencies. If a line contains a. formula. then its contest is the set of a.11 declarations needed to make sense of that, formula, and parsing of the commands leading into it.

The dependencies are established using rules similar to Gentzen's Natural Deduction System.

- A line resulting from a. command assume depends on itself.

- A line resulting from a command define or trw inherits the dependencies of the lines quoted by the rewriter plus the lines that are used automatically, having previously been labeled as simpinf 0.

- A line resulting from a command rw inherits the dependencies of the lines quoted in line. rewriter and those labeled simpinf o.

- A line resulting from a. command derive inherits the dependencies of the lines quoted in linerange, rewriter and those labeled simpinf o.

- The dependencies of the line $\mathbf{line}_0$ resulting from a. command cases are determined as follows. Suppose the formula of the line is $A_1 \vee \ldots \vee A_n$, and suppose linerange is $\mathbf{line}_1 \ldots \mathrm{line}_n$: then the dependencies of $\mathbf{line}_0$ are the union, for $j = 1, \ldots, n$, of the dependencies of the line, that are different from A_j.

- The dependencies of the line $\mathbf{line}_0$ resulting from a. command CI are determined as follows. Let all the formulas in linerange result from the command assume. Then $\mathbf{line}_0$ inherits the dependencies of line and of rewriter, except for those inherited from linerange.

- A line resulting from a. command ue inherits the dependencies of linedg and rewriter.

A variable occurring in a. line is *implicitly universally quantified* if it does not occur free in any of the dependencies of the line in question. This condition corresponds to the restriction on the application of $\forall$ -introduction in Natural Deduction System. As noted above, implicitly universally quantified variables behave exactly as explicitly universally quantified variables: in particular, the ue command applies to them. We cannot allow implicitly universally quantified variables in lines coming from the axiom or def ax command. EKL must regard an axiom as creating dependencies. although it is instructed to be silent about them. Carefully writing a.11 the universal quantifiers in the axioms saves many unpleasant surprises to the user. The variable defined by define or def ax is not implicitly universally quantified: it, is to be regarded as the eigenvariable of an $\exists$ -elimination in Natural Deduction.

1.2.4. Controlling the Rewriting Process.

Certain substitutions are automatically performed by EKL in rewriting. For instance:

— if A and B differ only in the name of the bound variables and the corresponding names have the same sort, then $A=B, A\equiv B$ and $A\supset B$ are simplified to TRUE

— $P\equiv TRUE$, P=TRUE, $P\wedge TRUE$, $TRUE\supset P$, $FALSE\vee P$, $\neg\neg P$, IF TRUE THEN P ELSE Q are all simplified to P. etc.

Other cases of standard rewriting are listed in the user's Manual.

Control over the rewriting process is one of the most important features of EKL. The commands to specify rewriters are described in the user's Manual. We recall only the ones most frequently used in this paper.

The command

(use linerange &*rest* options)

tells EKL that all lines in linerange are to be applied to the term being rewritten, in the order given by linerange. A line is 'applied' to a?. term as follows.

-EKL identifies terms that differ only for the names of bound variables of the same sort. Let A be the term being rewritten: if the formula. of the line is A, then A is replaced by TRUE: if the formula of the line is $\neg A$ then A is replaced by FALSE.

—If the formula of the line is a. conjunction, both conjuncts are successively applied to the term being rewritten.

-EKL performs 'conditional rewriting': if the formula of the line is $B \supset A$, then the term A is replaced by TRUE, provided that the decision procedure derives B from the current contest.

-If the formula of the line is universally quantified, then instances of the formula?, the bound variables being replaced by suitable terms, are applied to the term.

--Suppose the formula. of the line is an equality of the form a=b and let the term being rewritten be a. formula containg a. If in the command the list. of options is empty, then the left member of the equality a is replaced by b in the formula. provided that b is 'simpler than a.

The notion of 'simplicity can be roughly described as follows. The espressions of the language of EKL are ordered lexicographically: we say that f is 'simpler' than g and a+b is 'simpler' than b+a. Moreover the expression f (x,f (x)) is 'simpler' than f (x , y) since it contain fewer basic symbols. The usual recursive definitions of terms from basic symbols and of propositions from atoinic'propositions give a. natural measure of complexity of the espressions: we say that f(x) is 'simpler' than f (x , f (x))

A list of options is available to make substitutions in other ways:

(i)	direction: reverse
(ii)	direction: simpler
(iii)	mode: exact
(iv)	mode: always
(v)	ue: ((var1 . terml) . . . (varj . termj))

By (i), we ask EKL to apply equalities in the reverse of the normal direction (replace in the term under consideration an occurrence of b, the right member of the equality, by a, the left member). or, by (ii), in whichever direction will make the formula simpler. By (iii), we ask EKL to make the substitution no matter whether the result will be simpler. without applying the line again to the terms produced by the first application, or. by (iv), applying the line as many times as possible. The option (v) allows us to apply the UE command to the line and then to apply the modified line to the term being rewritten.

We can restrict. the range of application of the line to parts of the term being rewritten by using the command

(part subpart &*rest* rewriter)

Loosely speaking, we can regard the set of parts of an expression as a. tree and denote a part by any label of the path that leads to it. For instance. the parts of

$$\forall x.p(a)\wedge(q(a)\vee r(x))$$

can be denoted as follows:

```
                          1
                 p(a)∧(q(a)∨r(x))

      1#1                                  1#2
      p(a)                            q(a)∨r(x)

                               1#2#1            1#2#2
                               q(a)             r(x)
```

Example:

```
1.  (assume |∀x.p(a)∧(q(a)∨r(x))|)

2.  (assume |a=b|)

3.  (rw 1 (use 2))
    ;∀X.P(A)∧(Q(A)∨R(X))

4.  (rw 1 (use 2 mode: exact))
    ;∀X.P(B)∧(Q(B)∨R(X))

5.  (rw 1 (part 1#2#1 (use 2 mode: exact)))
    ;∀X.P(A)∧(Q(B)∨R(X))
```

The command

open *&rest* symbols)

is equivalent to

(use linerange mode: exact),

where linerange consists of all the lines involved in the definition of the symbols in the list symbols.

We may want to call the decision procedure to rewrite a subformula of a line to TRUE. This is done by the command

(der *&rest* linerange)

Finally, we may use several rewriters within a. single command.

1.2.5. EKL and Natural Deduction.

A derivation in Gentzen-style Natural Deduction can be extracted from any EKL proof (although most of the time we don't see it).

Let us disregard the fact that EKL lines may result from declarations, i.e. that EKL proofs contain also some language specifications and are, in this respect, similar to Martin-Löf-style derivations.

Some commands of EKL corresponds to rules of Natural Deduction systems:

assume	assume
	$\wedge$ introduction
	$\wedge$ elimination
	$\vee$ introduction
cases	$\vee$ elimination
ci	$\supset$ introduction
...	$\supset$ elimination
...	$\forall$ introduction
ue	$\forall$ elimination
...	$\exists$ introduction
define	$\exists$ elimination

The missing rules are replaced by the derive command. Commands for the quantifiers include higher order quantification.

If we want to write EKL proofs in terms of Natural Deduction. we must also include some form of equational calculus corresponding to the rewriting process. EKL does not display all the steps of substitutions in the process of rewriting. It. displays only the result of such process. We can ask EKL to show us all of the steps executed while rewriting by typing the command

```
(setq rewritemessages t)
```

(examples are given in Sections 2.1 ,2.6 and 2.9). Each step of rewriting corresponds to an application of a rule of equality in equational calculus. The rewriting of a nontrivial line may involve a huge number of substitutions. It is clear, then. why we do not want always to see the natural deduction derivation corresponding to an EKL proof.

More generally, to simulate the flexibility of informal reasoning through (mechanical simulation of) formal reasoning is an important aim in the field of automatic theorem proving. The details of the formalization of informal arguments may be ignored once we are convinced that the mechanical procedure is correct.

Since the rewriting process applies to logical simplification as well, we can replace applications of natural deduction rules with rewriting. In other words, we tend to apply rules of substitution and of replacement. perhaps repeatedly in a. single command, instead of expanding the proof according to the rules of natural deduction. This makes the EKL proofs much shorter. We shall show later some useful techniques to help the rewriting process and derive lines in one step.

1.2.6. **Remembering Lines in EKL.**

> *Forgetting is no mere* vis inertiae *as the superficial imagine: it is rather an active and in the strictest sense positive faculty of repression... The man in whom this apparatus of repression is damaged and ceases to function properly may be compared ... with a dyspeptic – he cannot 'have done' with anything.* †

EKL is capable of remembering and forgetting. The command

(label name *&optional* linerange)

tags the lines in linerange with label name. Linerange defaults to the last line of the current proof.

(unlabel name *&optional* linerange)

removes the label name from the tags associated to each line. Linerange defaults to the last line of the current proof.

A state in EKL consists of the currently active proof. the currently active context. the currently active linename contest and the currently active rewritename contest.

A *linename context* is a. list of symbolic names associated to lines. These associations may be set by the LABEL command.

A rewritename context is a. list of symbolic names associated to rewriters. These associations may be set by the REWRITENAME command.

The label simpinfo has special meaning to the rewriter. The lines labeled simpinf o are assumed to be lines that are always used in rewriting for simplification purposes or for verifying sorts.

We can call lines not only by name, but also by their number. The command

(use f oo#3)

means: use the third line of the proof f oo. The command

(use -3)

means: use the third line in the current, proof before the one being written. The symbol * stands for -1, i.e., it denotes the last line.

The *currently active context* is the cumulative subtotal of all the contest manipulation that has happened in the currently active proof.

h a. typical command several lines may be citecl. We first of all combine the contests of the cited lines. If an incompatibility turns up, the command is aborted. This contest is then combined with the previous active contest; all the incompatible declarations from the previous contest are thrown out. The resulting contest is then used for parsing of terms etc. in the command. If no contest lines are cited, we default to the previous contest. This is sufficient most of the time.

It follows that we can use conflicting declarations in different parts of the of the same proof provided that we do not try to refer to these lines within the same command: the language that is used is ultimately local to the line in question.

† F.Nietzsche, *Genealogy of Morals*, Second Essay, in: Kaufmann (editor) *Basic Writings of Nietzsche*, pag.493-4.

1.3. Rudiments of LISP.

We shall use lists to represent, finite functions. Let us quickly recall the basic notions of LISP. (The following may also be regarded as a. commentary to the file LISPAX. containing the Axioms of LISP. to be found in the Appendix.)

Given a set $\mathcal{A}$ of atoms, including the empty list NIL. the set $\mathcal{S}$ of symbolic expressions (S-expressions), is the set built from the atoms using the pairing operation "•":

(i) ACS

(ii) if x and y are S-expression then x . y is an S-expression.

In other words,

$$\mathcal{S} = \mathcal{A} + \mathcal{S} \times \mathcal{S}.$$

The unary operations car and cdr are the first and the second projections, defined on $\mathcal{S} \setminus \mathcal{A}$. It is convenient for our purpose to define

```
car nil = nil = cdr nil.
```

The set. $\mathcal{L}$ of lists is a. subset of the set. $\mathcal{S}$ of S-expressions. $\mathcal{L}$ is defined inductively by the clauses

(iii) NIL is a list.

(iv) if u is a list and x is an S-espression then x . u is a list.

As usual, we abbreviate $(a_1 . (a_2 . . . . . (a_n$. NIL)...)) as $(a_1\, a_2 \, . . . \, a_n)$. The variables xa. ya and za always range over S-espressions (i.e. are of sort atom)? x, y and z range over S-expressions (sort sexp) and u. v, w range over lists (sort listp).

These inductive definitions suggest principles to define functions by recursion on the definitions of $\mathcal{S}$ (*recursion on S-expressions*) and $\mathcal{L}$ (*recursion on lists*). Using higher order logic we can formulate the principle *Listinductiondef* of recursion on lists as

```
∀df nilcase def.
   (∃fun.(∀pars x u.fun(nil,pars)=nilcase(pars)∧
                    fun(x.u,pars)=def(x,u,fun(u,df(x,pars)),pars)))
```

Here pars is a list of n parameters, df is a. given auxiliary (72 + 1)-ary function, giving a list of n parameters as value, `nilcase` is a given n-ary function and def is a given n •t 3-ary function. for each n. Actually the type structure of EKL plays a major role here, since it. can be used to transform any list of n arguments into a single argument. For esample, fun is declared to have type

```
ground⊗ground*→ground*.
```

We can also formulate the principle of *Listinduction* to prove facts about funct ions defined by recursion on $\mathcal{L}$:

```
∀phi.phi(nil)∧(∀x u.phi(u)⊃phi(x.u))⊃(∀u.phi(u))
```

Here phi is any predicate taking lists as argument. The priniciples of recursion and induction on S-expressions are similar.

The type structure of the language of EKL is a. limit to the inductive strengt h of t he syst em. In the sit uat ion described a bove

-pars is of type ground*.

-df of type (ground⊗ground*)→ground*,

—nilcase of type ground*→ground*,

so that fun will be of type (ground⊗ground*)→ground*, too.

The device of variable types is a way to overcome such limitation. Consider the following *High Order Definition*

```
Vbigfun atom_fun.∃defined_fun.
  ∀x y. (atom x ⊃ defined_fun(x)=atom_fun(x))∧
       (defined_fun(x.y)=
          bigfun(x,y,defined_fun(x),defined_fun(y)))
```

Here

-arb is a variable type with name ?arbitrary,

-bi,gf un is of type ground⊗ground⊗@arb⊗@arb→@arb,

-def ined_fun and atom-fun are of type ground→@arb.

In this way we allow EKL to postpone the decision about the type of the function def ined_f un to the time of application of the principle to define a particular function iii a given context: then arb can be specialized to an object of *any* type. Therefore we have a primitive recursive schema for definit ion on *all* higher type functionals.

1.4. Permutations and the Pigeon Hole Principle.

Let A be a finite set and let 3 be the set of a.11 *surjections* on A, i.e. the set of all functions mapping A *onto* itself. The following fact is an easy consequence of the Pigeon Hole Principle.

Lemma. *Every surjection on a finite set is an injection.*

The proof will be considered in section below . Assuming the Lemma, it is not hard to prove the following Theorem.

Theorem. (3. o), *where o is the operation of composition of functions, is a group.*

Proof. It is easy to check that the composition of two surjections on A is a surjection on A and that composition of functions is an associative operation. The identity map i is a surjection and is the two-sided identity with respect to o. Finally, given $f \in \mathcal{F}$, the inverse map

$$f^{-1} : f(a) \mapsto a,$$

for all $a \in A$, is a well defined function. since f is an injection: f^{-1} has A as domain, since f is a surjection on A; f^{-1} is a surjection on A, since the domain of f is A. Clearly, for all $f \in$ 3.

$$f^{-1} \circ f = i = f \circ f^{-1}. \quad \blacksquare^{\dagger}$$

† We use ■ for the end of a proof (both informal and mechanical) and □ for the end of an example.

The pigeon hole principle is usually formulated as follows: if we put $n+1$ pigeons in n holes, then at least one hole gets more than one pigeon. Equivalently,

If we have n *pigeons and* n *holes and each hole contains at least one pigeon, then each hole contains exactly one pigeon.*

More formally, let $\mathbf{N}_n$ be the segment of N bound by n, i.e. the set of natural numbers less than n.

Theorem. *Let* f *be a function on natural numbers,* $f : \mathbf{N}_n \rightarrow N$, *such that for all* m,

$$(i) \qquad f(m) > 0$$

and

$$(ii) \qquad \sum_{m=0}^{n-1} f(m) = n.$$

Then for all m<n,

$$f(m) = 1.$$

Proof. We use induction on n, employing the following facts of arithmetic: for all k, m, n,

$$(iii) \qquad m \geq n \wedge k \geq 1 \supset m + k \geq n + 1,$$

for all k, m, n,

$$(iv) \qquad m \geq n \wedge k \geq 1 \wedge m + k = n + 1 \supset m = n \wedge k = 1.$$

We use *(i)* and *(iii)* to show, by induction on n, that for all n,

$$(v) \qquad \sum_{m=0}^{n-1} f(m) \geq n.$$

Now, in the induction step, we assume

$$\sum_{m=0}^{n} f(m) = n + 1.$$

and use (iv) and (v) to prove

$$(vi) \qquad \sum_{m=0}^{n-1} f(m) = n \wedge f(n) = 1;$$

hen we apply the induction hypothesis. ■

Now we can prove

Lemma. *Every surjection on a finite set is an injection.*

Proof. Let $|A| = n$ be the cardinality of A and let a_m be the m-th element, for some enumeration without repetition of A. For any sequence of pairwise disjoint sets B_i,

$$(vii) \qquad \sum_{i<n} |B_i| = \left|\bigcup_{i<n} B_i\right|.$$

Let $f \in 3$ and for each $m < n$, let A,, = $f^{-1}(\{a_m\})$, the inverse image of $\{a_m\}$. The A_i's are pairwise disjoint and their union is A. Therefore, by (vii),

$$(viii) \qquad \sum_{m<n} |A_m| = |A| = n.$$

Moreover, since f is surjective, for all A, and all $m < n$,

$$(ix) \qquad |A_m| > 0.$$

The Pigeon Hole Principle says that, if $(viii)$ and (ix), then for all $m < n$,

$$(x) \qquad |A_m| = 1.$$

We conclude, for all i, j $< n$,

$$a_i \neq a_j \supset f(a_i) \neq f(a_j),$$

by applying (m) to $A_{f(i)}$. ■

1.5. The Representation of Permutations in LISP.

We turn now to the representation of finite functions in terms of of LISP structures and the operations on finite functions as LISP programs. We will consider the representation of the above mathematical facts as properties of LISP programs and formally state the facts to be proved by EKL.

1.5.1. A Remark on Sets and Lists.

A set, according to Cantor's explanation, is an aggregate of objects, regarded as an entity that can itself be an element of other sets. In Set Theory sets may be constructed out of a. given stock of basic objects. the *urelements*, but abstraction is made from the particular features of the urelements as well as from the order in which the urelements may be given to us. (In fact. in mainstream Set Theory urelements are ignored and the entire universe of sets is generated out. of nothing, from the empty set.)

In formalizing Set Theory within, say, first order logic, a distinction is made between sets and classes in order to avoid paradoxes: unlike sets, classes cannot be regarded as elements of other sets or classes and axioms (say, Zermelo-Frankel asioms) determine if a. property. expressed by a predicate, actually denotes a set or only a. class.

If the formal language is a typed language, as the language of EKL, we may disregard the distinction between sets and classes, for the strict, restriction imposed by the type structure already guarantee from paradoxes. Thus instead of

$$\{x : P(x)\}$$

we may write (using the lambda notation)

$$\lambda x.P(x)$$

or simply

$$P$$

to denote the set of objects having the property P. The ϵ relation can then be defined as

Definition. (*Epsilon*)

```
∀av xv.xv∈av≡av(xv)
```

We will use the epsilon notation applied only to the relation between urelements and sets of urelements.

The set

$$\{x\}$$

can be represented as

$$\lambda y.y = x.$$

This is our notation for the singleton set:

Definition. (*Mkset*)

```
∀xv.mkset(xv)=(λyv.yv=xv).
```

Given an aggregate, if we abstract only from the particular features of the elements we have an *ordered set;* if the set is finite we speak of a *list.* In the LISP language the term 'list' has a technical meaning, and membership in a list is represented by the recursive predicate member.

Definition. (*Member*)

```
∀x y u. ¬member(x,nil)∧member(x,y.u)≡(x=y∨member(x,u))
```

Conceptually, the distinction between a list u and the set

$$\{x : x \text{ is a member of u}\} \qquad (*)$$

amounts to the distinction between a finite ordered set and a set. Our notation for the set $(*)$ is

Definition. (*Mklset*)

```
∀u.mklset(u)=λx.member(x,u)
```

The functional mklset maps a lists into the set of its members.

1.5.2. **Permutations as Association Lists.**

Let $f : A \rightarrow B$ be any finite function. i.e. a. function defined on a finite set A. Its graph, i.e. the set $\{(a, f(a)) : a \in A\}$, can be written as

$$\begin{pmatrix} a_1 & a_2 & & a_n \\ f(a_1) & f(a_2) & \cdots & f(a_n) \end{pmatrix}. \qquad (\star)$$

A 'finite function can be represented by an association list, i.e. by writing the graph of the function as a list. For instance the above function f can be represented by the list $\mathtt{alist}_f$

$$(\ (a_1 \cdot f(a_1))\ (a_2 \cdot f(a_2))\ ...\ (a_n \cdot f(a_n))\ ).$$

The notation $(\star)$ is slightly ambiguous: the graph of a function is a *set* of pairs, but $(\star)$ seems rather to denote a list of pairs. Strictly speaking, the graph of f is correctly represented by $\mathtt{mklset(alist}_f\mathtt{)}$, not by $\mathtt{alist}_f$. It is more informative to represent f by an equivalence class of association lists rather than by a predicate. We give an appropriate equivalence relation below.

Using $\mathtt{alist}_f$, we can represent the operation 'apply f to an element a_k in the domain of f' as follows: take the cdr of the S-expression in $\mathtt{alist}_f$ whose car is a_k. Moreover, if f and g are finite functions such that the composition g o f of f and g is defined and $\mathtt{alist}_f$ and $\mathtt{alist}_g$ are the association lists representing f and g, then we can find a list $\mathtt{alist}_{g \circ f}$

$$(\ (a_1 \cdot \ldots \cdot c_2)\ \ldots \cdot c_n)\ )$$

representing g 0 f as follows: "given a_k, in order to find c_k go through $\mathtt{alist}_f$ searching for the S-expression, whose car is a_k and take its cdr, say b_k; nest. go through $\mathtt{alist}_g$ searching for the S-expression, **whose car** is b_k; and take its cdr as c_k".

The identity and inverse operations have an easy representation using association lists. The following list $\mathtt{alist}_{id}$ represents the identity function on $\{a_1, \ldots . a_n\}$:

$$(\ (a_1 \cdot a_1)\ (a_2 \cdot a_2)\ ...\ (a_n \cdot a_n)\ ).$$

The 'inverse' of $\mathtt{alist}_f$ is given by the list $\mathtt{alist}_{f^{-1}}$:

$$(\ (f(a_1) \cdot a_1)\ (f(a_2) \cdot a_2)\ ...\ (f(a_n) \cdot a_n)\ )$$

The result of 'composing $\mathtt{alist}_{f^{-1}}$ with $\mathtt{alist}_f$ is $\mathtt{alist}_{id}$. The result of 'composing' alistf with $\mathtt{alist}_{f^{-1}}$ is the following list $\mathtt{alist}_{id1}$:

$$(\ (f(a_1) \cdot f(a_1))\ (f(a_2) \cdot f(a_2))\ ...\ (f(a_n) \cdot f(a_n))\ )$$

If f is a bijection (and $\mathtt{alist}_{id}$, $\mathtt{alist}_{id1}$ "don't contain garbage") then both $\mathtt{alist}_{id}$ and $\mathtt{alist}_{id1}$ represent the identity function on the same set.

The official EKL definition of `alist` is given by the following axiom. Here `alist` is a variable of type ground and sort. alistp.

Definition. (*Alistdef)*

```
∀xa y alist.alistp nil A alistp (xa.y).alist
```

The following is the definition of the operation of application using association lists.

Definition. (*Appalist*)

```
∀alist y.appalist(y,alist)=cdr assoc(y,alist)
```

where assoc has been defined in the LISP library file as follows:

```
∀x xa y alist.assoc(x,nil)=nil∧
          assoc(x,(xa.y).alist)=(if x=xa
                                   then xa.y
                                   else assoc(x,alist))
```

Given an association list alist, let dom(alist) be the list containing the first element of each pair and range(alist) the list of all the second elements.

Definition. (*Dom*)

```
∀xa y alist.dom nil=nil∧
        dom((xa.y).alist)=xa.dom alist
```

Definition. (*Range*)

```
∀xa y alist.range nil=nil∧
        range((xa.y).alist)=y.range alist
```

The recursive predicate uniqueness is true of a list u iff every element of u occurs only once.

Definition. (*Uniqueness*)

```
∀u x.uniqueness nil ∧
   (uniqueness(x.u)≡¬member(x,u)∧uniqueness(u))
```

The fact that (the equivalence class of) alist represents a function is given by the property of *uniqueness* of dom(alist):

Definition. (*Functp*)

```
∀alist.functp(alist)≡uniqueness dom(alist)
```

and the fact that (the equivalence class of) alist represents an injection can be characterized as follows:

Definition. (*Injectp*)

```
∀alist.injectp(alist)≡functp(alist)∧uniqueness range(alist)
```

Finally, (the equivalence class of) alist represents a permutation if and only if dom(alist) and range(alist) are the same *as sets* and have the same length *as lists*. The second property is of course obviously true for any alist.

Definition. (*Permutp*)

```
∀alist.permutp(alist)≡functp(alist)∧
                 mklset(dom(alist))=mklset(range(alist))
```

We don't need to include in our definition of permutation the fact that (the equivalence class of) **alist** represents an injection, namely injectp(alist). The fact that the property **injectp** follows from our definition permutp corresponds to the Lemma in Section 1.4.

Composition of functions is then represented **by** the **following LISP** function ∘ (notice the order — $alist_f$ ∘ $alist_g$ is the function $g \circ f$):

Definition. (*Compalist*)

```
∀alist1 alist2 xa y.nil ∘ alist2=nil∧
                   ((xa.y).alist1) ∘ alist2=
                        (xa.appalist(y,alist2)).(alist1 ∘ alist2)
```

Identity is represented by the following predicate:

Definition. (*Idalistp*)

```
∀alist xa y . idalistp(nil)∧
             (idalistp((xa.y) .alist)≡xa=y∧idalistp alist)
```

Inversion is represented by the **LISP** function:

Definition. (*Invalist*)

```
∀alist xa y . invalist nil=nil∧
             invalist((xa.y).alist)=(y.xa).invalist alist)
```

An unpleasant feature of this approach is that any association list consisting exactly of the S-espressions $(a_k \,.\, f(a_k))$, for all k, is also a representation of f, independently of the order in which they occur. The function f is not represented by a single association list, but by the class of all association lists that have the same members and give the same result with respect to the operation of "application".

The equivalence relation is represented by the following predicate:

Definition. (*Samemap*)

```
∀alist alist1.samemap(alist,alist1)≡
          mklset dom(alist)=mklset dom(alist1)∧
          (∀y.y∈mklset dom(alist)⊃
               appalist(y,alist)=appalist(y,alist1))
```

The Theorem to be proved consists of the following statements:

Theorem 1. (i) (*Permutp Compalist*)

```
∀ALIST ALIST1.PERMUTP(ALIST)∧PERMUTP(ALIST1)∧
          MKLSET(DOM(ALIST))=MKLSET(DOM(ALIST1))⊃
          PERMUTP(ALIST ∘ ALIST1)
```

Theorem 1. (ii) (*Compalist Associativity*)

```
∀ALIST ALIST1 ALIST2.MKLSET(RANGE(ALIST))⊂MKLSET(DOM(ALIST1))⊃
                 ALIST ∘ (ALIST1 ∘ ALIST2)=(ALIST ∘ ALIST1) ∘ ALIST2
```

Theorem 2. (i) (*Idalistp Permutp*)

```
∀ALIST.FUNCTP(ALIST)∧IDALISTP(ALIST)⊃PERMUTP(ALIST)
```

Theorem 2. (ii) (Right *Idalistp*)

```
∀ALIST1.IDALISTP(ALIST1)⊃
 (∀ALIST.MKLSET(RANGE(ALIST))⊂MKLSET(DOM(ALIST1))⊃ALIST ∞ ALIST1=ALIST)
```

Theorem 2. (iii) (*Left Idalistp*)

```
∀ALISTID ALIST.IDALISTP(ALISTID)∧
            MKLSET(DOM(ALISTID))=MKLSET(DOM(ALIST))⊃
            SAMEMAP(ALISTID ∞ ALIST,ALIST)
```

Theorem 3. (i) (*Permutp Invalist*)

```
∀ALIST.PERMUTP(ALIST)⊃PERMUTP(INVALIST(ALIST))
```

Theorem 3. (ii) (*Right Invalist*)

```
∀ALIST.ALLP(λX.ATOM X,RANGE(ALIST))∧INJECTP(ALIST)⊃
       IDALISTP(ALIST ∞ INVALIST(ALIST))
```

Theorem 3. (iii) (*Left Invalist*)

```
∀ALIST.ALLP(λX.ATOM X,RANGE(ALIST))∧INJECTP(ALIST)⊃
       IDALISTP(INVALIST(ALIST) ∞ ALIST)
```

1.5.3. **Permutations as Lists of Numbers.**

Let $\mathbf{N}_n$ be the segment of N up to n, i.e. the set $\{m : m \in \mathbf{N} . m < n\}$. If A is the set $\mathbf{N}_n$ and f is a function with domain A then f is called a (finite) sequence.

We can represent arbitrary finite functions using finite sequences. Given $f : A \to B$ and suitable bijections $i : \mathbf{N}_n \to A$, $j : \mathbf{N}_m \to B$ where n is the cardinality of A and m is the cardinality of the range of f, there is a finite function $g : \mathbf{N}_n \to \mathbf{N}_m$ such that the diagram

$$\begin{tikzcd} A \arrow[r, "f"] & B \\ \mathbf{N}_n \arrow[u, "i"] \arrow[r, "g"] & \mathbf{N}_m \arrow[u, "j"'] \end{tikzcd}$$

commutes. Thus, we need only consider functions from segments of N to segments of N.

Although lists and finite sequences are essentially the same kind of mathematical object , a function is usually understood as a *method* to associate an element of the range to each element of the domain in a unique way.

When finite functions are represented by lists. we specify a method as follows. Given the finite function $h : N_n \to N$. "list the range" of h in the order given by the domain, i.e. construct the list $\mathbf{v}_h$

$$(h(0)\ h(1)\ \ldots\ h(n-1)).$$

Thus h associates to each number in $\mathbf{N}_n$ the *nth* element of $\mathbf{v}_h$. (To "list the domain" in the order given by the range is another possibility.)

The LISP function nth is defined as follows:

Definition. (*Nth*)

```
∀x u n.nth(nil,n)=nil∧nth(u,0)=car u∧nth(x.u,n')=nth(u,n)
```

The equation

$$\mathtt{nth}(\mathbf{v}_h, k) = h(k)$$

explains how the function nth represents the operation of applying a. function to a. number.

If $\mathbf{v}_h$ represents h, and u is any list of numbers, then $\mathbf{v}_h$ can be "applied" to u, by applying $\mathbf{v}_h$ successively to all the members of u. The operation "applying $\mathbf{v}_h$" to u is defined if all members of u are numbers less than the length of $\mathbf{v}_h$.

This motivates our official definition of application. using lists of numbers:

Definition. (*Appl*)

```
∀u i.appl(u,i)=nth(u,i)
```

The following predicate specifies the condition for v to be defined as an application on u as the domain:

```
∀u v.def_appl(v,u)≡allp(λx.natnum(x)∧x<length(v),u)
```

Here allp is a recursive predicate, checking whether all members of a list have a certain property:

```
∀phi x u.allp(phi,nil)∧
         allp(phi,x.u)=if phi(x) then allp(phi,u) else false
```

The fact that a list u represents an injection is naturally represented by the predicate inj: if every element of u occurs just once in u, then two applications of u give the same value only for the same argument.

Definition. (*Inj*)

```
∀u.inj(u)=∀n m.n<length(u)∧m<length(u)∧nth(u,n)=nth(u,m)⊃n=m
```

On the other hand. the fact that u represents a surjection on $\mathbf{N}_{length(u)}$ is given by the property onto, namely the fact that all members of u are numbers in $\mathbf{N}_{length(u)}$ and, conversely. all numbers in $\mathbf{N}_{length(u)}$ are members of u. In such case every number in $\mathbf{N}_{length(u)}$ will be the result of an application of u to some argument.

Definition. (Onto)

```
∀u.into(u)=(∀n.n<length u⊃natnum nth(u,n)∧nth(u,n)<length u)

∀u.onto(u)=(into(u)∧(∀n.n<length u⊃member(n,u)))
```

Definition. (*Perm*)

```
∀u.perm(u)=onto(u)
```

As above. we don't need to include in our definition of permutation the fact that f is 1-1 : the proof' that `perm(u)` implies inj (u) will be described in Section 1.5.4.

Composition of functions can be represented by the following LISP function:

Definition. (*Compose*)

```
∀u v x.(u⊗nil)=nil∧(u⊗(x.v))=(nth(u,x)).(u⊗v)
```

Equivalently. the following predicate `comp` gives the condition for an application of u to be the same as an application of w followed by an application of v.

Definition. (*Comp*)

```
∀u v w.comp(u,v,w)≡
      length u=length w∧(∀n.n<length u⊃nth(u,n)=nth(v,nth(w,n)))
```

The representation of the identity function and the inversion of permutations are discussed in Sect ion 6.1. It is clear that the predicate id gives the condition for the result of an application of u to be the same as its argument:

Definition. (Id)

```
∀u.id(u)≡(∀n.n<length u⊃nth(u,n)=n)
```

We will choose the following function to construct the list representing the identity function:

Definition. (*Ident*)

```
∀x u n i.ident1(i,0)=nil∧ident1(i,n')=i.ident1(i',n)

∀n.ident(n)=ident1(0,n)
```

Consider the function λu x .f `stposition(u,x)` that ret urns a. number n, with $0 \leq n <$ -length(u). corresponding to the position of the first, occurrence of x in u. if x occurs in u. and NIL otherwise.

```
∀x u y.fstposition(nil,y)=nil∧
       fstposition(x.u,y)=if ¬member(y,x.u)
                             then nil
                             else if x=y
                                     then 0
                                     else add1(fstposition(u,y))
```

This funct ion is our candidate for the inverse operation of nth. If x occurs in u. then

```
nth(u,fstposition(u,x))=x.
```

By applying this for x = nth(u,n) and n < length(u). we get

```
fstposition(u,nth(u,n))=m,
```

with m < length(u). Here m need not be equal to n. However, this will certainly be the case if x occurs only once in u, or in other words if u has the *Injectivity* property inj (u).

Notice the asymmetry here: the function fstposition is the *right* inverse of λn.nth(u,n) for *any* u_h, i.e. for any function *12* represented by u_h. However fstposition is the *left* inverse of Xn. nth(u ,n) only if u_h has the *injectivity* property. i.e. if the function h represented by u_h is a permutation.

Using this property of fstposition, we can give the condition for u to represent the inverse function of the permutation v:

Definition. (*Inv*)

```
Vu v.inv(u,v)≡(∀n.n<length u⊃nth(u,n)=fstposition(v,n))
```

and, as argued below. the following is a. convenient way of constructing such inverse:

Definition. (*Inverse*)

```
Vu i n.invers1(u,i,0)=nil∧invers1(nil,i,n)=nil∧
    invers1(u,i,n')=if null(fstposition(u,i))
                    then nil
                    else fstposition(u,i) . inversl(u,i',n)

∀u.inverse(u)=invers1(u,0,length(u))
```

Using predicates, the results to be proved are:

Theorem 1. (i) (*Composition*)

```
VU V W.PERM(V)∧PERM(W)∧LENGTH V=LENGTH W∧COMP(U,V,W)⊃PERM(U)
```

Theorem 1. (ii)(*Uniqueness*)

```
∀U U1 V W.COMP(U,V,W)∧COMP(U1,V,W)⊃U=U1
```

Theorem 1. (iii)(*Associativity*)

```
vu U1 v V1 W1 w2 w3.
   INTO(W3)∧LENGTH W2=LENGTH W3∧
   COMP(V,W1,W2)∧COMP(U,V,W3)∧COMP(V1,W2,W3)∧COMP(U1,W1,V1)⊃U=U1
```

Theorem 2. (i)(*Identity*)

```
∀U.ID(U)⊃PERM(U)

VU V W.ID(U)∧COMP(V,W,U)∧LENGTH W=LENGTH U⊃V=W
```

Theorem *2.* (ii)(*Right Identity)*

VU V W.ID(U)∧PERM(W)∧LENGTH W=LENGTH U∧COMP(V,W,U)⊃W=V

Theorem 2. (iii)(*Left Identity*)

∀U V W.ID(U)∧PERM(W)∧LENGTH W=LENGTH U∧COMP(V,U,W)⊃W=V

Theorem 3. (i)(*Inverse*)

VU V.PERM(U)∧INV(V,U)∧LENGTH V=LENGTH U⊃PERM(V)

Theorem 3. (ii)(*Right Inverse*)

VU V W.PERM(W)∧INV(U,W)∧COMP(V,W,U)∧LENGTH U=LENGTH W⊃ID(V)

Theorem 3. (iii)(*Left Inverse*)

VU V W.PERM(W)∧INV(U,W)∧COMP(V,U,W)∧LENGTH W=LENGTH U⊃ID(V)

Using functions, the results can be staked as follows:

Theorem 1. (i) (*Perm Compose*)

VU V.PERM U ∧ PERM V ∧ LENGTH U = LENGTH V ⊃ PERM(U⊗V)

Theorem 1. (ii) (*Associativity of Composition*)

VU V W.PERM(V)∧PERM(U)∧LENGTH V=LENGTH U∧LENGTH W=LENGTH U⊃
(W⊗V)⊗U=W⊗(V⊗U)

Theorem 2. (i)(*Perm Ident*)

∀N.PERM(IDENT(N))

Theorem 2. (ii)(*Right Identity)*

∀U.U⊗IDENT(LENGTH U)=U

Theorem 2. (iii)(*Left Identity*)

∀U.INTO(U)⊃IDENT(LENGTH U)⊗U=U

Theorem 3. (i)(*Perm Inverse*)

∀U.PERM(U)⊃PERM(INVERSE(U))

Theorem 3. (ii)(*Right Inverse*)

∀U.PERM(U)⊃U⊗INVERSE(U)=IDENT(LENGTH(U))

Theorem 3. (iii)(*Left Inverse*)

∀U.PERM(U)⊃INVERSE U⊗U=IDENT(LENGTH U)

1.5.4. Application of the Pigeon Hole Principle to Permutations.

We have two representations of finite functions: thus we will have prove two facts representing the theorem that every finite surjection is an injection. In the representation by **alists** the fact is:
Theorem (*Permutp Injectp*)

```
∀ALIST.PERMUTP(ALIST)⊃INJECTP(ALIST)
```

Under the assumption **permutp(alist)**, we need to show uniqueness range **(alist)**.
In the representation by lists of numbers we show:

Theorem (*Perm Injectivity*) ∀U.PERM(U)⊃INJ(U)

As explained above. *uniqueness* and *injectivity* are equivalent predicates, asserting that every element of a list occurs just once. Although the theorems in question can be formulated in terms of the definition of permutation and of the predicates above, we need more information when we try to prove them.

The argument for theorem *Permutp Injectp* can be summarized as follows. Since by definition dom(**alist**) has the uniqueness property. there are n different *kinds* of objects (n 'holes') in dom(alist) and also in **range(alist)**, since **dom(alist)** and **range(alist)** have objects of the same kinds (i.e., each 'hole has at least one object ('pigeon') of **range(alist)**). The number of (distinct) objects in range **alist** ('pigeons') is at most the length of **range(alist)** and at least the number of different kinds of objects. therefore it is exactly n. Therefore each kind of object occurs just once in **range(alist)** and this implies that range **alist** has the uniqueness property.

Despite the apparent triviality of this informal argument, some work is needed to formalize it. To speak of 'kinds' of objects is to speak of sets. We need a function counting the multiplicity of elements of u belonging to the set a:

Definition (*Multiplicity):*

```
∀x u a.mult(nil,a)=0∧
       mult(x.u,a)=if a(x) then mult(u,a)' else mult(u,a)
```

Next we must show that the list **dom(alist)**, considered as a set. can be partitioned into disjoint. sets, i.e. the sets

$$A, = \{x: x=\text{nth}(\text{dom } \mathbf{alist}, n)\}$$

for all n, $n <$ **length(dom(alist))**.

Therefore we need a recursive predicate to decide whether the sets of a sequence are pairwise disjoint:

Definition (*Disjoint*):

```
∀n setseq.
    disjoint(setseq,0)∧
    disjoint(setseq,n')=(disjoint (setseq,n)∧
                          disj_pair(un(setseq,n),setseq(n)))
```

where **disj_pair** is defined as

```
∀a b.disj_pair(a,b)=emptyp(a∩b)
```

To count the distinct objects in range(alist) we need the notions of finite union and finite sum:

Definition (*Finite Union*):

```
∀n setseq.un(setseq,0)=emptyset∧
          un(setseq,n')=un(setseq,n)∪setseq(n)
```

Definition (*Finite Sum*):

```
∀n numseq.sum(numseq,0)=0∧
          sum(numseq,n')=sum(numseq,n)+numseq(n)
```

and, moreover, the following

Lemma (*Mult of Un is Sum Mult*)

```
∀SETSEQ U N.DISJOINT(SETSEQ,N)⊃
           MULT(U,UN(SETSEQ,N))=SUM(λX1.MULT(U,SETSEQ(X1)),N)
```

The argument for Theorem *Perm Inj* is similar, but simpler. As before we prove the rather obvious fact. that $\mathbf{N}_n$ can be partitioned into the disjoint sets

$$\{x : x = m\}$$

for each $m <$ length(u). We need to show that for each $m <$ length(u) the multiplicity in u of the set {x: x = m}, is exactly 1; then the injectivity of u follows. The pigeon-hole principle is used to prove this fact.

The pigeon-hole principle as such is an easy matter also for EKL. We use simple numeric induction to prove that for any function $f : \mathbf{N}_n \rightarrow \mathrm{N}$ if the values of f are at least 1 and the sum of n values is n then each value is exactly 1. In both applications the function in question is

```
λm.mult(v,a(m))
```

In the case of association lists, v is `range(alist)`; in the other case it is the given list u. In the case of association lists, a(m) is the set {x: x=nth(dom **alist ,m)**}; in the case of numeric lists we can take the set {x : x = m} for a(m) and this is the reason why in this case proofs arc simpler.

1.6. Outline of the Paper.

All the proofs are given in the Appendix. The organization of proofs in files and the dependence of the files arc described at the beginning of the Appendis.

Part I, i.e. Sections 2 and 3 can be regarded as an introductory guide to automatic deduction of facts about LISP. through experiments and examples.

Section 2 is devoted to the definition of the LISP functions nth, nthcdr. fstposition and mult and to the proof of basic facts about them. It also contains facts of set theory and arithmetic.

Some useful techniques to replace "deriving" by "rewriting" through tautologies of second order propositional logic are esplained and illustrated with an example.

We prove. among other things, the following facts connecting member and nth...

Lemma 2.1. (*Nth Member*)

VU N.N<LENGTH U⊃MEMBER(NTH(U,N),U)

Lemma 2.2. (*Member Nth*)

VU Y.MEMBER(Y,U)⊃(∃N.N<LENGTH U∧NTH(U,N)=Y)

. ..and the following properties of nthcdr:

Lemma 2.3. (*Nthcdr Car Cdr*)

VU N.N<LENGTH U⊃NTHCDR(U,N)=NTH(U,N).NTHCDR(U,N')

Lemma 2.4. (*Nth in Nthcdr*)

VU N M.N≤M∧M<LENGTH U⊃MEMBER(NTH(U,M),NTHCDR(U,N))

Facts about nth and fstposition:

Lemma *2.5. (Nth Fstyosition)*

VU N.MEMBER(N,U)⊃NTH(U,FSTPOSITION(U,N))=N

Lemma *2.6. (Fstyosition Nth)*

VU N.UNIQUENESS(U)∧N<LENGTH U⊃FSTPOSITION(U,NTH(U,N))=N

The *set* of elements of a. list is the finite union of the sets obtained using nth:

Lemma 2.7. (*Mklset Un*)

∀U.UN(λM.MKSET(NTH(U,M)),LENGTH(U))=(λX.(MKLSET(U))(X))

Moreover we show the following facts concerning the function **mult**:

Lemma *2.8. (Length Mult)*

VU A.MULT(U,A)≤LENGTH U

Lemma 2.9. (*Member Mult*)

VU Y A.MEMBER(Y,U)∧A(Y)⊃1≤MULT(U,A)

Lemma 2.10. (*Mult Nthcdr*)

∀N A U.N<LENGTH U⊃MULT(NTHCDR(U,N),A)≤MULT(U,A)

Lemma 2.11. (*Mult Inj*)

∀V.(∀K.K<LENGTH V⊃MULT(V,MKSET(NTH(V,K)))=1)⊃INJ(V)

The following facts about finite sums and unions are also needed:

Lemma 2.12. (*Multsum*)

∀U.DISJ_PAIR(A,B)⊃MULT(U,A∪B)=MULT(U,A)+MULT(U,B)

Lemma 2.13. (*Mult of Un is Sum Mult*)

```
VSETSEQ U N.DISJOINT(SETSEQ,N)⊃
            MULT(U,UN(SETSEQ,N))=SUM(λX1.MULT(U,SETSEQ(X1)),N)
```

Section 3 contains the definitions of *application* and *permutation.* in both representations. It contains also some facts needed for the representation through association lists. In particular, since this representation is not unique, we have the predicate **samemap** that is true of two alists if they represent, the same function. We show that **samemap** is an equivalence relation on al ist s:

Lemma *3.1.* (*Samemap Equivalence*)

```
(i)   ∀ALIST.SAMEMAP(ALIST,ALIST)

(ii)  ∀ALIST ALIST1.SAMEMAP(ALIST,ALIST1)⊃SAMEMAP(ALIST1,ALIST)

(iii) ∀ALIST ALIST1 ALIST2.SAMEMAP(ALIST,ALIST1)∧SAMEMAP(ALIST1,ALIST2)⊃
                           SAMEMAP(ALIST,ALIST2)
```

Part 2 contains the three mathematical facts, namely

— the proof of the Pigeon Hole Principle, and two proofs that every finite surjection is an injection *(Section 4)*;

— the proof that permutations represented as association lists form a group *(Section 5)*;

— the proof that permutations represented as lists of numbers form a. group *(Section 6)*.

Section 4. contains

Theorem. (*Pigeonfact*)

```
∀F N.(∀M.M≤N⊃NATNUM F(M))∧(∀M.M<N⊃1≤F(M))∧SUM(λK.F(K),N)≤N⊃
    (∀M.M<N⊃1=F(K))
```

Corollary. (*Pigeonlist*)

```
∀U.DISJOINT(SETSEQ,LENGTH U)⊃
   ((∀M.M<LENGTH U⊃1≤MULT(U,SETSEQ(M)))⊃
   (∀M.M<LENGTH U⊃1=MULT(U,SETSEQ(M))))
```

and the two applications of the pigeon hole principle. In both cases the proof takes three steps.

In the version representing functions as association lists the desired result....

Theorem. (*Permutp Injectp*)

```
∀ALIST.PERMUTP(ALIST)⊃INJECTP(ALIST)
```

. ..is proved through the following steps:

Lemma 4.1. (*Inj Disj*)

```
∀U.INJ(U)⊃DISJOINT(λM.MKSET(NTH(U,M)),LENGTH U)
```

Lemma *4.2.* (*Permutp Injectp Lemma)*

```
∀U V.MKLSET(U)=MKLSET(V)⊃
             (∀M.M<LENGTH(U)⊃1≤MULT(V,MKSET NTH(U,M)))
```

Lemma 4.3. (*Mult Mult*)

```
VU V.MKLSET(U)=MKLSET(V)∧
     (∀M.M<LENGTH U⊃MULT(V,MKSET(NTH(U,M)))=1)⊃
     (∀I.I<LENGTH V⊃MULT(V,MKSET(NTH(V,I)))=1)
```

The conclusion follows by the lemma *Mult Inj.*

In the version representing functions as lists the result...

Theorem. (*Perm Injectivity*)

```
∀U.PERM(U)⊃INJ(U)
```

. ..is proved again in three steps:

Lemma *4.4. (Disjoint Number*)

```
∀N.DISJOINT(λXV.MKSET(XV),N)
```

Lemma *4.5.* (*Onto* ***Mult***)

```
∀U.ONTO(U)⊃
   (∀N.N<LENGTH(U)⊃1≤MULT(U,MKSET(N)))
```

Lemma *4.6.* ***(Into*** *Mult*)

```
∀U.INTO(U)∧
   (∀K.K<LENGTH U⊃1=MULT(U,MKSET(K)))⊃
   (∀I.I<LENGTH U⊃1=MULT(U,MKSET(NTH(U,I))))
```

The conclusion follows using the lemma *Mult Inj.*

Sections 5 – *6* contain definitions of the operation *composition* of functions, of the *identity* function and of the operation taking the *inverse* of a permutation and proofs of the following theorems:

Theorem 1. (i) *The composition of permutations is a permutation.*

(ii) *Composition of functions is associative.*

Theorem ***2.*** (i) *The identity function i is a permutation.*

(ii) *For every permutation* f, f o i = f.

(iii) *For every permutation* f, i o f = f.

Theorem *3.* (i) *For every permutation* f*, the inverse function* f^{-1} *is a permutation.*

(ii) *For every permutation f,* f o f^{-1} = i.

(iii) *For every permutation f,* f^{-1} o f = i.

In **Section *5*** we work with association lists. In the proof of the theorems we need the following facts:

Lemma 5.1 (*App Compalist*)

```
VALIST ALIST1.MEMBER(X,DOM(ALIST))⊃
 APPALIST(X,ALIST m ALIST1)=APPALIST(APPALIST(X,ALIST),ALIST1)
```

Lemma 5.2 (*Dom Compalist*)

VALIST ALIST1.DOM(ALIST m ALISTl)=DOM(ALIST)

Lemma 5.3 (*Nonempty Range*)

VALIST X.MEMBER(X,DOM ALIST)⊃
(∃Y.MEMBER(Y,RANGE ALIST)∧APPALIST(X,ALIST)=Y)

Lemma 5.4 *(Nonempty Domain)*

VALIST Z.UNIQUENESS DOM(ALIST)∧MEMBER(Z,RANGE ALIST)⊃
(∃X.MEMBER(X,DOM ALIST)∧APPALIST(X,ALIST)=Z)

Lemma 5.5 (*Main Idalistp*)

VALIST Y.IDALISTP(ALIST)∧MEMBER(Y,DOM(ALIST))⊃
APPALIST(Y,ALIST)=Y

In **Section 6** first we discuss the choice of LISP functions and predicates for the representation through lists of numbers. Then the proofs of the theorems in the representations PERMP. using predicates, and PERMF. using LISP functions. are shown in parallel.

In the version PERMF we need first to prove some facts about length.

Lemma 6.1. (*Length* Compose)

VU W.DEF_APPL(W,U)⊃LENGTH (W⊗U)=LENGTH U

Lemma 6.2. (*Length Ident*)

VN.LENGTH (IDENT(N))=N

Lemma 6.3. *(Length Inverse)*

∀U.PERM(U)⊃LENGTH (INVERSE(U))=LENGTH U

In the version PERMF by proving first the following facts. we make it possible to follow the proofs of the version PERMP.

Lemma 6.4. *(Nth Compose)*

VU N.DEF_APPL(V,U)∧N<LENGTH U⊃NTH(V⊗U,N)=NTH(V,NTH(U,N))

Lemma 6.5. *(Main Id)*

∀N.N<M⊃NTH(IDENT(M),N)=N

Lemma 6.6. (*Main Inv*)

VU N.PERM U∧N<LENGTH U⊃NTH(INVERSE U,N)=FSTPOSITION(U,N)

PART 1

2. Preliminaries: Basic Tools.

2.1. Educating EKL about propositional Logic.

One of the unique features of EKL is the ability to describe procedures like bringing formulas into disjunctive normal form (where other rewriters can then be applied) as a set of simple rewriters. However, since this is often not appropriate and may cause combinatorial explosions. we do not add these to the default rewrite facts denoted by simpinfo; instead we want to call those lines as rewriters when needed.

```
;propositional schemata, used by the rewriter to normalize expressions
(proof normal)

1   (trw |∀p q r.((p∨q)∧r)≡((p∧r)∨(q∧r))|)
    (label normal)

2   (trw |∀p q r.(r∧(p∨q))≡((r∧p)∨(r∧q)) |)
    (label normal)

3   (trw |∀p q r . ((p∨q)∧r)≡((p∧r)∨(q∧r))|)
    (label normal )

4   (trw |∀p q r. (p∨q⊃r)≡(p⊃r)∧(q⊃r) |)
    (label normal )

5   (trw |∀p q.(¬(p∨q))≡((¬p)∧(¬q))|)
    (label demorgan)

6   (derive |∀p q.¬(p∧q)≡(¬p)∨(¬q)|)
    (label demorgan1)
```

Now the rewriter will be able to normalize espressions, distributing conjunction over disjunction, eliminating disjunctions in the antecedent of an implication and negations of disjunctions.

The pure rewriter. however, finds it difficult to make certain inferences in conditional rewriting. This problem may be overcome by introducing propositional facts to be used later as rewriters.

```
7.  (derive |∀p q.p≡(q⊃p)∧(¬q⊃p)|)
    (label excluded-middle)

8.  (derive |∀p q r.(q⊃r)∧(if p then q else r)⊃r|)
    (label trans_cond)
```

Remark. Example 1. The use of the lines labeled NORMAL is an interesting example of use of second order unification. Since sentences are just terms of type truthval, we can apply to them the rewriting procedure in a uniform way. This is made possible, of course. by the use of the higher order unification. We give an example of its application. The fact to prove is the transitivity of $\leq$, assuming the transitivity of $<$. Using our technique we collapse into one line a 16 line long Natural Deduction style proof'. We will present the Natural Deduction Style proof first.

```
(wipe-out)
(get-proofs nth prf prm glb)
(proof example)

(setq rewritemessages t)

;labels: TRANSITIVITY-OF-ORDER
;∀N M K.N<M∧M<K⊃N<K

;labels: LESSEQDEF
;∀M N.M≤N=(M=N∨M<N)
```

(Remember: (open lesseq) is the same as use: lesseqdef mode: exact.)

```
0.  (trw |∀n m k.n≤m∧m≤k⊃n≤k| (open lesseq)
        transitivity-of-order)
    ;the term N≤M is replaced by:
    N=M∨N<M
    ;the term M≤K is replaced by:
    M=K∨M<K
    ;the term N≤K is replaced by:
    N=K∨N<K
    ;(∀N M K.N≤M∧M≤K⊃N≤K)≡(∀N M K.(N=M∨N<M)∧(M=K∨M<K)⊃N=K∨N<K)
```

We do not go very far by simply expanding the definition of $\leq$, because the rewriter does not know what to do with the disjunctions in the antecedent.

Instead. we can construct a derivation and use two arguments by cases to handle the disjunctions (lines 1-I and 15).

```
    (setq rewritemessages nil)

1.  (assume |n≤m|)
    (label example1)

2.  (assume |m≤k|)
    (label example2)

3.  (rw example1 (open lesseq))
    ;N=M∨N<M
    (label example3)
    ;deps: (EXAMPLE1)
```

Argue by cases. First case:

```
4.  (assume |n=m|)

5.  (rw example2 (use * mode: exact direction: reverse>>
    ;N≤K
    (label example4)
```

Second case:

```
6.   (assume |n<m|)
     (label example5)

7.   (rw example2 (open lesseq))
     ;M=K∨M<K
     (label example6)
     ;deps: (EXAMPLE2)
```

Within the second case, we need another argument by cases.

```
8.   (assume |m=k|)

9.   (rw example5 (use * mode: exact))
     ;N<K

10.  (trw |n≤k| (open lesseq) *)
     ;N≤K
     (label example7)

11.  (assume |m<k|)

12.  (derive |n<k| (transitivity-of-order example5 *))

13.  (trw |n≤k| (open lesseq) *)
     ;N≤K
     (label example8)

14.  (cases example6 example7 example8)
     ;N≤K
     (label example10)
     ;deps: (EXAMPLE2 EXAMPLES)
```

This concludes the second case. So we can conclude our first argument.

```
15.  (cases example3 example4 example10)
     ;N≤K
     ;deps: (EXAMPLE1 EXAMPLE2)

16.  (ci (example1 example2))
     ;N≤M∧M≤K⊃N≤K
```

. This concludes the Natural Deduction style proof of the transitivity of $\leq$. However. using the rewriter NORMAL we can do a.11 this in one step.

```
0.   (trw |∀n m k.n≤m∧m≤k⊃n≤k| (open lesseq) (use normal mode: always)
          transitivity-of-order)
     ;∀N M K.N≤M∧M≤K⊃N≤K
     (label example)
```

For after expanding the definition of $\leq$ the rewriter uses lines 1 and 2 of the proof NORMAL

```
;the term (N=M∨N<M)∧(M=K∨M<K) is replaced by:
N=M∧(M=K∨M<K)∨N<M∧(M=K∨M<K)
;the term N=M∧(M=K∨M<K) is replaced by:
N=M∧M=K∨N=M∧M<K
```

So in the first disjunct

```
;the term M is replaced by:
K
```

Similarly

```
;the term N<M∧(M=K∨M<K) is replaced by:
N<M∧M=K∨N<M∧M<K
;the term M is replaced by:
K
```

Later the rewriter uses line 4 of the proof NORMAL. This corresponds to argument by cases.

```
;the term N=K∧M=K∨N=M∧M<K∨N<K∧M=K∨N<M∧M<K⊃N=K∨N<K is replaced by:
(N=K∧M=K⊃N=K∨N<K)∧(N=M∧M<K∨N<K∧M=K∨N<M∧M<K⊃N=K∨N<K)
```

Now standard rewriting does the job for the first conjunct:

```
;the term N=K is replaced by:
TRUE
;the term TRUE∨N<K is replaced by:
TRUE
;the term N=K∧M=K⊃TRUE is replaced by:
TRUE
```

etc. □ †

2.2. Educationg EKL about first grade Arithmetic.

First we ask EKL to read the proofs contained file MINUS. namely the proofs "minus" and "lesseq". They in turn contain the instruction of reading the files NATNUM and NORMAL (see the Appendis).

```
(wipe-out)
;Done.Proof?

(get-proofs minus)
;file read in
;switched to MINUS
;the proof LESSEQ read in.
;the proof INDUCTION read in.
;the proof MINUS read in.
;the proof NATNUM read in.
;the proof NORMAL read in.
```

† We use □ for the end of an example and ■ for the end of a proof (both informal and mechanical).

2.3. LISP and the Bound Quantifier Allp.

Similarly we ask EKL to learn about LISP by reading the file LISPAX (see the Appendix).

```
(wipe-out)
;Done.Proof?
(get-proofs lispax)
;file read in
;switched to LISPAX
;the proof LISPAX read in.
```

In defining function&, the language of EKL gives us the option between a definition by recursion and a definition using bounded quantifiers.

Consider the predicate allp(phi ,u), to be interpreted as "for all members x of u. phi(u). It could be defined as:

```
(define allp |∀phi x u.allp(phi,u)≡(∀x.member(x,u)⊃phi(x))|)
```

The definition by recursion *Allpdef*

```
∀phi x u.allp(phi,nil)∧
          allp(phi,x.u)=if phi(x) then allp(phi,u) else false
```

simplifies its use in proofs by induction on lists: consider for instance the proof's of the Lemma *Nth Compose* or of Theorem Assoc Compose. In contests where a. straightforward proof by induction is not possible. we may use the other definition. having proved the equivalence.

```
;facts about allp
(proof allp)

;a reformulation of the definition of allp

1    (trw |∀phi x u.allp(phi,x.u)⊃phi(x)∧allp(phi,u)| (open allp))
     ;∀PHI X U.ALLP(PHI,X.U)⊃PHI(X)∧ALLP(PHI,U)
     (label allpfact)

     ;allp_introduction

2    (ue (phi |λu.(∀y.member(y,u)⊃phi1(y))⊃allp(phi1,u) |>
         listinduction
         (open allp member) (use normal mode: always))
     (label allp,introduction)
     ;∀U.(∀Y.MEMBER(Y,U)⊃PHI1(Y))⊃ALLP(PHI1,U)

     ; allp-elimination

3    (ue (phi |λu.member(x,u)∧allp(phi1,u)⊃phi1(x)|)
         list induction
         (part 1 (open member allp) (use normal mode: always)))
     (label allp-elimination)
     ;∀U.MEMBER(X,U)∧ALLP(PHI1,U)⊃PHI1(X)

     ;allp_implication
```

```
4.  (ue (phi |λu.∀a a1.allp(a,u)∧(∀x.a(x)⊃a1(x))⊃allp(a1,u)|)
        listinduction (open allp) )
    (label allp_implication)
    ;∀U A A1.ALLP(A,U)∧(∀X.A(X)⊃A1(X))⊃ALLP(A1,U)
```

Similarly for the predicate somep:

```
    Vphi x u.¬somep(phi,nil)∧
          somep(phi,x.u)=if phi(x) then true else somep(phi,u)

    (proof someprop)

1   (ue (phi |λu.member(y,u)∧phi1(y)⊃somep(phi1,u)|)
        listinduction
        (open somep member) (use normal mode: always))
    ;∀U.MEMBER(Y,U)∧PHI1(Y)⊃SOMEP(PHI1,U)

2   (derive |∀u.(∃y.member(y,u)∧phi1(y))⊃somep(phi1,u)| *)

3   (ue (phi |λu.somep(phi1,u)⊃(∃x.member(x,u)∧phi1(x))|)
        listinduction
        (part 1 (open member somep) (use normal mode: always) (der)))
    ;∀U.SOMEP(PHI1,U)⊃(∃X.MEMBER(X,U)∧PHI1(X))

4   (derive |∀u.somep(phi1,u)≡(∃x.member(x,u)∧phi1(x))| (* -2))
    (label somepf act )
```

2.4. Facts of elementary set theory.

Next we introduce some useful notations of elementary set theory. We do not distinguish between sets and predicates: our variables **av,** bv for sets will allow us to speak only about very few sets (only sets of "urelements", sets of objects of type ground-see the file 1.5.1).

Remark. Example 2. The following example shows that some care is needed in dealing with default declarations. In guessing the declaration for a term, EKL looks for syntactical similarities with previously defined terms: thus if x has been previously declared, EKL tries the same declaration for xl or xv.

If we start a new proof, without access to the previous ones, then the expression xv receives default declaration type: ground syntype : variable sort: universal.

```
    (proof sets)

I.  (decl (av bv) (type: |ground→truthval|))
2.  (decl epsilon (type: |ground⊗@av→truthval|)
          (infixname: ∈) (bindingpower: 925))

3.  (define epsilon |∀av xv.xv∈av≡av(xv)|)
    ;XV is unknown.
    ;the symbol XV declared to have type GROUND
```

On the other hand, in the proof **LISPAX** the term x has already been declared: its declaration is type: ground syntype: variable sort : sexp. Therefore, if we give EKL access to the proof **lispax** first, then xv becomes a variable of the sort sexp (line 3 below).

Since in this paper we will consider only sets of S-espressions. such default declaration is convenient.

```
(get-proofs allp)
;file read in
;switched to ALLP
; the proof ALLP read in.
;the proof LISPAX read in.

(proof sets)

1. (decl (av bv) (type: |ground→truthval|))
2. (decl epsilon (type: |ground⊗@av→truthval|)
         (infixname : ∊) (bindingpower: 925))

3. (define epsilon |∀av xv.xv∊av≡av(xv)|)
   (label epsilondef)
   ;XV is unknown.
   ;the symbol XV is given the same declaration as X
```

However, there is a. more elegant way to obtain this result: we can declare xv to be of some sort, Say urelement:

```
(wipe-out)
(proof set)

(decl (xv yv zv) (type: ground) (sort: urelement))
```

Then we establish, by axioms, that urelements and S-expressions are the same class.

```
(axiom |∀x.urelement x|)
(label simpinfo)

(axiom |∀xv.sexp xv|)
(label simpinfo)
```

Thus we can create the two files separately and later give EKL access to both files and assume the above axioms, if needed. □

```
;useful set theory
(wipe-out)
(get-proofs allp)
(proof sets)

;all urelements will be S-expressions
;all S-expressions will be urelements

1. (decl (xv yv zv) (type: lgroundl) (sort: urelement))
2. (decl (av bv) (type: |ground→truthval|))
```

```
3.  (axiom |∀x.urelement x|)
    (label simpinfo)

4.  (axiom |∀xv.sexp(xv)|)
    (label simpinfo)

5.  (decl epsilon (type: |ground⊗@av→truthval|)
          (infixname: ∈) (bindingpower: 925))
6.  (define epsilon |∀av xv.xv∈av≡av(xv)|)
    (label epsilondef)
    ;XV is unknown.
    ;the symbol XV is given the same declaration as X

7.  ;∀A B.(∀XV.XV∈A≡XV∈B)⊃A=B
    (label set,extensionality)

8.  (decl intersection (type: |@set⊗@set→@set|)
          (infixname: ∩) (bindingpower: 950)
          (prefixname: intersection) )
9.  (define intersection |∀a b.a∩b=λxv.(a(xv)∧b(xv))|)
    (label interdef)

10. (decl union (type: |@set⊗@set→@set|)
          (infixname: ∪) (bindingpower: 950)
          (prefixname: union))
11. (define union |∀a b.a∪b=λxv.(a(xv)∨b(xv))|)
    (label uniondef )

12. (decl inclusion (type: |@set⊗@set→truthval|)
          (infixname: ⊂) (bindingpower: 920)
          (prefixname: inclusion) )
13. (define inclusion |∀a b.a⊂b≡∀xv.a(xv)⊃b(xv)|)
    (label inclusiondef )

14. (defax emptyset |emptyset=λxv.false|)
    (label emptysetdef)

15. (defax emptyp |∀a.emptyp(a)=∀xv.¬a(xv)|)
```

We want to be able to talk of the set of occurrences of an S-expressions x as well as of the set of elements of a list u.

```
16. (decl mkset (type: |ground→@set|))

17. (define mkset |∀xv.mkset(xv)=(λyv.yv=xv)|)
    (label mkset,def)

    ;the set of members of a list

18. (decl mklset (type: |ground→@av|))
19. (define mklset |∀u.mklset(u)=λx.member(x,u)|)
    (label mklsetdef)
```

2.5. Putting things together.

The basic ground domain will contain both S-expressions and natural numbers. We need both to define the function length

```
∀u x.(length nil=0)∧length(x.u)=(length u)'
```

(see the Appendix).

```
(get-proofs length)
;file read in
;switched to SETFACTS
;the proof SETFACTS read in.
;the proof ALLP read in.
;the proof LESSEQ read in.
;the proof INDUCTION read in.
;the proof LENGTH read in.
;the proof MINUS read in.
;the proof NATNUM read in.
;the proof NORMAL read in.
;the proof SETS read in.
;the proof LISPAX read in.
```

In such context, the following principle (*Doubleinduction1*) of double induction for lists and numbers will be very useful:

```
∀PHI3.
(∀U N X.PHI3(NIL,N)∧PHI3(U,0)∧(PHI3(U,N)⊃PHI3(X.U,N')))⊃(∀U N.PHI3(U,N))
```

Numbers and S-espressions are ground objects of different sorts.

```
(axiom |∀n.sexp n|)
(label simpinfo)

(axiom |∀n.¬null(n)|)
(label simpinfo)
```

We remarked above that. some care is needed to give the database the proper structure of types and sorts. In our experiment, no artificial limitation of expressive power is imposed by the type structure of EKL. Now we are ready to introduce the main LISP functions needed for our representations of permutations.

2.6. Properties of Nth.

The LISP function nth plays a. key role in our represent ation. nth and nthcdr are defined as total functions, with the default value NIL. We shall present. facts about these functions as examples of simple inferences in EKL.

```
(proof nth)

1.  (decl nth (syntype: constant) (type: |ground⊗ground→ground|))

2.  (defax nth |∀x u n.nth(nil,n)=nil∧nth(u,0)=car u∧
                      nth(x.u,n')=nth(u,n)|)
    (label simpinfo) (label nthdef)
```

Example 3. The well-definedness of nth is an immediate consequence of its definition by double induction on lists and numbers. We show the rewriting process in detail. Without the use of simpinfo the following statement is obtained.

```
    (setq rewritemessages t)

0.  (ue (phi3 |λu n.sexp nth(u,n)|) doubleinductionl (nuse simpinfo))
    ;(∀U N X.SEXP NTH(NIL,N)∧SEXP NTH(U,O)∧
    ;        (SEXP NTH(U,N)⊃SEXP NTH(X.U,N')))⊃
    ;(∀U N.SEXP NTH(U,N))
```

The information in simpinfo, including the definition of nth, is enough to obtain the result.

```
3.  (ue (phi3 |λu n.sexp nth(u,n)|) doubleinductionl)
    ;the term NTH(NIL,N) is replaced by:
    NIL
    ;the term SEXP NIL is replaced by:
    TRUE
    ;the term SEXP NTH(U,O) is replaced by:
    TRUE
    ;the term NTH(X.U,N') is replaced by:
    NTH(U,N)
    ;the term SEXP NTH(U,N) is replaced by:
    TRUE
    ;the term SEXP NTH(U,N)⊃TRUE is replaced by:
    TRUE
    ;the term TRUE∧TRUE∧TRUE is replaced by:
    TRUE
    ;the term W N X.TRUE is replaced by:
    TRUE
    ;the term TRUE⊃(∀U N.SEXP NTH(U,N)) is replaced by:
    W N.SEXP NTH(U,N)
    ;∀U N.SEXP NTH(U,N)
    (label simpinfo) (label sexp_nth)      ■
```

□

Lemma 2.1. (*Nth Member*)

```
W N.N<LENGTH U⊃MEMBER(NTH(U,N),U)
```

Proof. We use double induction also the membership of the values of nth in the original list. The first base case, when $n = 0$, is proved by listinduction. For u = NIL we obtain a contradiction in the antecedent (the line ZEROLEAST1, proof NATNUM, is in simpinfo). For u = x.u we apply definitions of nth and of member.

```
(ue (phi |λu.0<length u⊃member(nth(u,0),u)|) listinduction
    (open member))
;∀U.0<LENGTH U⊃MEMBER(NTH(U,0),U)
```

The other base case gives again a. contradiction and the inductive step is immediately reduced to the induction hypothesis. Indeed, n'<length(x.u) reduces to n'<(length(u))' and by SUCCESSORLESS (proof NATNUM) to n<length(u). By definition, nth(x.u,n') = nth(u,n).

```
(ue (phi3 |λu n.n<length u ⊃ member(nth(u,n),u) |) doubleinductionl
    (use memberdef mode: always) (use * ))
; W  N.N<LENGTH U⊃MEMBER(NTH(U,N),U)
(label nthmember)
```
∎

We need a converse of NTHMEMBER:

Lemma 2.2(*Member Nth*)

W Y.MEMBER(Y,U)⊃(∃N.N<LENGTH U∧NTH(U,N)=Y)

Proof. Since *Member Nth* is an existential statement, we have to expand the proof. We use induction on the list u. In order to prove that

∃n.n<length(x.u)∧nth(x.u,n)=y,

assume the induction hypothesis (line 1) and the antecedent for the inductive step (line 2), (line 11).

```
     (proof member,nth)

1.   (assume |(MEMBER(Y,U)⊃(∃N.N<LENGTH U∧NTH(U,N)=Y))|)
     (label m_n1)
     ;deps: (1)

2.   (assume |member(y,x.u)| )
     (label m_n2)
     ;deps: (2)

3.   (rw * (open member))
     (label m_n3)
     ;Y=X∨MEMBER(Y,U)
     ;deps: (M_N2)
```

This requires a proof by cases.

```
4.   (assume |y=x|)
     (label m_n4)
     ;deps: (4)
```

If $y = x$, one can take 0 for the desired n. It is enough to expand the definitions of length and nth in line 5 to verify that

```
0<length(x.u)∧nth(x.u,0)=y.
```

```
5.  (trw |0<length(x.u)∧nth(x.u,0)=y| * )
    ;0<LENGTH (X.U)∧NTH(X.U,0)=Y
    ;deps: (M_N4)

6.  (derive |∃n.n<length(x.u)∧nth(x.u,n)=y| * )
    (label m_n5)
    ;deps: (M_N4)
```

Second case:

```
7.  (assume |member(y,u)|)
    (label m_n6)
    ;deps: (7)

8.  (define nv |nv<length u∧nth(u,nv)=y| (m_n1 *))
    ;NV is unknown.
    ;the symbol NV is given the same declaration as N
    ;deps: (M_N1 M_N6)
```

The command DEFINE allows the introduction of an eigenvariable. This is EKL's way to deal with existential elimination. Now take nv' for n:

```
9.  (trw |nv'<length(x.u)∧nth(x.u,nv')=y| * )
    ;NV'<LENGTH (X.U)∧NTH(X.U,NV')=Y
    ;deps: (M_N1 M_N6)

10. (derive |∃n.n<length(x.u)∧nth(x.u,n)=y| * )
    (label m_n7)
    ;deps: (M_N1 M_N6)
```

Existential introduction is performed in lines 6 and 10 by the DERIVE command. In both cases we have reached the desired conclusion.

```
11. (cases m_n3 m_n5 m_n7)
    ;∃N.N<LENGTH (X.U)∧NTH(X.U,N)=Y
    ;deps: (M_N1 M_N2)
```

Cases derives the formula of lines 6 and 10 (the formula must be the same) and discharges the open assumptions of lines 4 and 7, respectively, by using line 3. We use conditional introduction to discharge assumptions and to write down the induction step (line 13). In line 14 the inductive argument is performed as a rewriting procedure, using line 13 as a rewriter.

```
12. (ci m_n2)
    ;MEMBER(Y,X.U)⊃(∃N.N<LENGTH U'∧NTH(X.U,N)=Y)
    ;deps: (M_N1)

13. (ci M_N1)
    ;(MEMBER(Y,U)⊃(∃N.N<LENGTH U∧NTH(U,N)=Y))⊃
    ;(MEMBER(Y,X.U)⊃(∃N.N<LENGTH U'∧NTH(X.U,N)=Y))
```

The base case is trivial. since NIL has no members. Therefore:

14. (ue (phi |λu.member(y,u)⊃(∃n.n<length u∧nth(u,n)=y)|) listinduction
(open member) *)
;∀U.MEMBER(Y,U)⊃(∃N.N<LENGTH U∧NTH(U,N)=Y) ■

2.7. Properties of Nthcdr.

(proof nthcdr)

1. (decl nthcdr (syntype: constant) (type: |ground⊗ground→ground|))

2. (defax nthcdr |∀x u n.nthcdr(nil,n)=nil∧nthcdr(u,0)=u∧
nthcdr(x.u,n')=nthcdr(u,n)|)
(label simpinfo) (label nthcdrdef)

The proofs of the following facts are quite **easy** and can be found in the Appendix.

3. VU N.LISTP NTHCDR(U,N)
(label simpinfo)

4. ∀U.0<LENGTH U⊃NTH(U,0).NTHCDR(U,1)=U
(label nth,nthcdr,zero)

5. VU N.N<LENGTH U⊃CAR NTHCDR(U,N)=NTH(U,N)
(label car,nthcdr)

6. VU N.CDR NTHCDR(U,N)=NTHCDR(U,N')
(label cdr_nthcdr)

Lemma 2.3. (*Nthcdr Car Cdr*)

7. VU N.N<LENGTH U⊃NTHCDR(U,N)=NTH(U,N).NTHCDR(U,N')
(label nthcdr,car,cdr) ■

The proof of the following Lemma is of some interest. We give it here.
Lemma 2.4. (*Nth in Nthcdr*)

VU N M.N≤M∧M<LENGTH U⊃MEMBER(NTH(U,M),NTHCDR(U,N))

Proof. First, we show

VU N M.(N<M∧M<LENGTH U⊃MEMBER(NTH(U,M),NTHCDR(U,N)))

by double induction on numbers and lists, i.e. on n and on u (line 13). For $n = 0$ the result is just the lemma *Nthmember*. For u = NIL we have a false antecedent.

As the inductive hypothesis we need an explicitly universally quantified formula.:

1. (assume |∀m.(n<m∧m<length u⊃member(nth(u,m),nthcdr(u,n)))|)
(label nincdrl)

The inductive step is proved by a. secondary induction on m. The case $m = 0$ gives a false antecedent. When m is a successor the inductive formula. is rewritten to an instance of line 10. using the definitions of nth, nthcdr, length and the fact *Successorless*. file NATNUM. which is in simpinfo.

Notice that we must tell EKL *not* to use the definitions of nth, nthcdr and length in the part of the formula. that corresponds to the conclusion.

```
2.  (ue (a |λm.(n'<m∧m<length(x.u)⊃member(nth(x.u,m),nthcdr(x.u,n')))|)
        proof-by-induction
        (part 2 (nuse nthdef nthcdrdef lengthdef))
        nincdrl  zero-non-less-successor)
    ;∀N2.N'<N2∧N2<LENGTH (X.U)⊃MEMBER(NTH(X.U,N2),NTHCDR(X.U,N'))

3.  (ci nincdrl)
    ;(∀M.N<M∧M<LENGTH U⊃MEMBER(NTH(U,M),NTHCDR(U,N)))⊃
    ;(∀N2.N'<N2∧N2<LENGTH U'⊃MEMBER(NTH(X.U,N2),NTHCDR(U,N)))
```

We can conclude the main induction.

```
4.  (ue (phi3 |λu n.∀m.n<m∧m<length(u)⊃member(nth(u,m),nthcdr(u,n))|)
        doubleinductionl
        (use nthmember mode: exact) (use * mode: exact))
    ;(∀U N M.N<M∧M<LENGTH U⊃MEMBER(NTH(U,M),NTHCDR(U,N)))
```

It is interesting to notice that the above argument can be replaced by a. one line proof. using proof-by-induction as a rewriter.

```
0.  (ue (phi3 |λu n.∀m.n<m∧m<length u⊃member(nth(u,m),nthcdr(u,n))|)
        doubleinductionl
        (use nthmember mode: exact)
        (use proof-by-induction
             ue: ((a.|λm.(n'<m∧m<length(u)'⊃
                          member(nth(x.u,m),nthcdr(u,n)))|))
             mode: exact))
    ;∀U N M.N<M∧M<LENGTH U⊃MEMBER(NTH(U,M),NTHCDR(U,N))
```

In the last step an argument by cases is avoided by our technique of using second order unification (line *Normal*).

```
5.  (trw |∀u n m.n≤m∧m<length(u)⊃member(nth(u,m),nthcdr(u,n))|
         (open lesseq member)(use normal mode: always)
         (use * nthcdr,car,cdr mode: exact))
    ;∀U N M.N≤M∧M<LENGTH U⊃MEMBER(NTH(U,M),NTHCDR(U,N))
    (label nth,in,nthcdr) ■
```

The proofs of the following facts are easy and left to the Appendis.

```
1.  VU N M.N<LENGTH U∧M<LENGTH(NTHCDR(U,N))⊃
            NTH(NTHCDR(U,N),M)=NTH(U,M+N)
    (label nth,nthcdr)

2.  ∀U N.N≤LENGTH U⊃LENGTH(NTHCDR(U,N))=LENGTH U-N
    (label length,nthcdr)

3.  ∀U.NTHCDR(U,LENGTH U)=NIL
    (label last,nthcdr)

4.  VU N.LENGTH(U)≤N⊃NTHCDR(U,N)=NIL
    (label trivial,nthcdr)

5.  VA U N.ALLP(A,U)⊃ALLP(A,NTHCDR(U,N))
    (label allp_nthcdr)
```

The principle of nthcdr induction can be viewed as a. trick to reduce induction on lists to finite induction on numbers. More interestingly, it is induction on lists *localized* to a given list, i.e. induction on the tails of a given list. Assume a list u is given; we can prove that u has a certain property phi from the fact that the null list has property phi and that if **x.v** is a tail of u and v has the property phi then **x.v** has the property phi. Using the functions nth and nthcdr we can formulate this method of proof as finite descent from phi (nthcdr (u, length(u))) to **phi(nthcdr(u,0))**.

The mechanical derivation of this inductive principle is not terribly interesting and is left to the Appendix.

```
6.  ∀PHI U.PHI(NIL)∧
          (∀N.N<LENGTH(U)⊃(PHI(NTHCDR(U,N'))⊃
              PHI(NTH(U,N).NTHCDR(U,N'))))⊃PHI(U)
    (label nthcdr,induct ion)
```

2.8. Properties of Fstposition.

In the representation of permutations the function f stposition plays the role of the inverse operation of nth. Here we give the definition of fstposition and some facts about it.

```
;fstposition
(proof fstposition)

1.  (decl (fstposition) (type: |ground⊗ground→ground|))
2.  (define f stposition
      |∀x u y.fstposition(nil,y)=nil∧
             fstposition(x.u,y)=if ¬member(y,x.u)
                                 then nil
                                 else if x=y
                                      then 0
                                      else addl(fstposition(u,y))|
      listinductiondef)
    (label fstpositiondef)
```

```
;facts about fstposition

3.  (ue (phi |λu.(null fstposition(u,y)⊃¬member(y,u))∧
              (member(y,u)⊃natnum fstposition(u,y))∧
              (null fstposition(u,y)∨natnum fstposition(u,y))|)
      listinduction
      (part 1 (open member fstposition) (use normal mode: always)))
    ;∀U.(NULL FSTPOSITION(U,Y)⊃¬MEMBER(Y,U))∧
    ;   (MEMBER(Y,U)⊃NATNUM(FSTPOSITION(U,Y)))∧
    ;   (NULL FSTPOSITION(U,Y)∨NATNUM(FSTPOSITION(U,Y)))
    (label simpinfo) (label posfacts)     ■

4.  (ue (phi |λu.∀y.sexp fstposition(u,y)|) listinduction
      (part 1 (open member fstposition) (use normal mode: always)))
    ;∀U Y.SEXP FSTPOSITION(U,Y)
    (label simpinfo) (label sortpos)     ■

5.  (ue (phi |λu.∀y.member(y,u)⊃fstposition(u,y)<length(u)|)
      listinduction
      (part 1 (open member fstposition) (use normal mode: always)))
    ;∀U Y.MEMBER(Y,U)⊃FSTPOSITION(U,Y)<LENGTH U
    (label pos,length) ■
```

2.9. The Lemmata Nth Fstposition and Fstposition Nth.

Since these facts are very basic, we comment the proofs in detail.

Lemma 2.5 (*Nth Fstposition*)

`∀U N.MEMBER(N,U)⊃NTH(U,FSTPOSITION(U,N))=N`

The proof that fstposition is the right inverse of nth is a simple induction on lists.

```
1.  (ue (phi |λu.∀n.member(n,u)⊃nth(u,fstposition(u,n))=n|)
      listinduction
      (use normal mode: always)
      (open member fstposition nth))
    ;∀U N.MEMBER(N,U)⊃NTH(U,FSTPOSITION(U,N))=N
    (label nth,fstposition) ■
```

To obtain the fact that fstposition is the left inverse of nth we need the additional hypothesis that u has the uniqueness property.

Lemma 2.6 (*Fstposition Nth*)

`∀U N.UNIQUENESS(U)∧N<LENGTH U⊃FSTPOSITION(U,NTH(U,N))=N`

Proof. By double induction on u and n.

(i) If u = NIL, then length(u) is 0, and we obtain a contradiction in the antecedent.

(ii) If n=0, we prove by induction on u that

$$\forall U.\mathrm{UNIQUENESS}(U)\wedge 0<\mathrm{LENGTH}\ U\supset \mathrm{FSTPOSITION}(U,\mathrm{CAR}\ U)=0.$$

The base case is like *i*, and the induction step is given by

$$\mathrm{fstposition}(x.u,\mathrm{car}(x.u)) = \mathrm{fstposition}(x.u,x) = 0.$$

(iii) Assume the induction hypothesis

$$\mathrm{uniqueness}(u)\wedge n<\mathrm{length}(u)\supset \mathrm{fstposition}(u,\mathrm{nth}(u,n))=n.$$

We want:

$$\mathrm{uniqueness}(x.u)\wedge n'<\mathrm{length}(x.u)\supset \mathrm{fstposition}(x.u,\mathrm{nth}(x.u,n'))=n'.$$

Assume uniqueness(x.u) and n'<length(x.u), which are rewritten as

$$\neg member(x,u)\wedge uniqueness(u) \qquad \text{and} \qquad n<length(u),$$

respectively.

(iv) Now

$$\mathrm{fstposition}(x.u,\mathrm{nth}(x.u,n'))$$

rewrites to

$$\text{if } x=\mathrm{nth}(u,n) \text{ then } 0 \text{ else } \mathrm{fstposition}(u,\mathrm{nth}(u,n))'.$$

We have only to show that x#nth(u,n): for then we can apply the induction hypothesis. But. if x=nth(u,n). with n<length u, then x is a. member of u, by *Nthmember*, contradicting (iii).

```
(proof fstposition-nth)

1. (ue (phi |λu.0<length u⊃fstposition(u,nth(u,0))=0|)
       listinduction (open fstposition nth member))
   ;∀U.0<LENGTH U⊃FSTPOSITION(U,CAR U)=0

2. (derive |n<length u ∧ x=nth(u,n) ⊃ member(x,u)| (nthmember))

3. (derive |uniqueness(x.u)∧n<length u⊃¬x=nth(u,n)| * (open uniqueness))

4. (ue (phi3 |λu n-uniqueness u∧n<length u⊃fstposition(u,nth(u,n))=n|)
       doubleinductionl *
       (open fstposition nth member uniqueness) -3 nthmember)
   ;∀U N.UNIQUENESS(U)∧N<LENGTH U⊃FSTPOSITION(U,NTH(U,N))=N
   (label f stposition,nth) ■
```

Remark. Example 4. The last line is a compact. proof. obtained by an interesting combination of rewriting steps. Let us look at the details of the rewriting process. The following statement must. be verified:

```
(setq rewritemessages t)

0.  (ue (phi3 |λu n.uniqueness u∧n<length u⊃fstposition(u,nth(u,n))=n|)
       doubleinductionl (nuse simpinfo))
    ;(∀U N X.(UNIQUENESS(NIL)∧N<LENGTH NIL⊃FSTPOSITION(NIL,NTH(NIL,N))=N)∧
    ;        (UNIQUENESS(U)∧0<LENGTH U⊃FSTPOSITION(U,NTH(U,0))=0)∧
    ;        ((UNIQUENESS(U)∧N<LENGTH U⊃FSTPOSITION(U,NTH(U,N))=N)⊃
    ;         (UNIQUENESS(X.U)∧N'<LENGTH (X.U)⊃
    ;          FSTPOSITION(X.U,NTH(X.U,N'))=N')))⊃
    ;(∀U N.UNIQUENESS(U)∧N<LENGTH U⊃FSTPOSITION(U,NTH(U,N))=N)
```

Using simpinf 0, without specifying any rewriter, only few substitutions are made. by the definition of nth and the fact *Successorless.*

```
0.  (ue (phi3 |λu n.uniqueness u∧n<length u⊃fstposition(u,nth(u,n))=n|)
       doubleinductionl)
    ;(∀U N X.(UNIQUENESS(U)∧0<LENGTH U⊃FSTPOSITION(U,CAR U)=0)∧
    ;        ((UNIQUENESS(U)∧N<LENGTH U⊃FSTPOSITION(U,NTH(U,N))=N)⊃
    ;         (UNIQUENESS(X.U)∧N<LENGTH U⊃FSTPOSITION(X.U,NTH(U,N))=N')))⊃
    ;(∀U N.UNIQUENESS(U)∧N<LENGTH U⊃FSTPOSITION(U,NTH(U,N))=N)
```

Let us see how the rewriting process simulates the above argument.

(i) First base case:

```
;the term UNIQUENESS(NIL) is replaced by:
TRUE
;the term LENGTH NIL is replaced by:
0
;the term N<0 is replaced by:
FALSE
;the term TRUE∧FALSE is replaced by:
FALSE
;the term FALSE⊃FSTPOSITION(NIL,NTH(NIL,N))=N is replaced by:
TRUE
```

Here EKL has found a. contradiction in the antecedent.

(ii) Nest EKL does the second base case, by espanding the definition of nth and using line 1:

```
;the term NTH(U,0) is replaced by:
CAR U
;the term FSTPOSITION(U,CAR U) is replaced by:
0
;the term 0=0 is replaced by:
TRUE
;the term UNIQUENESS(U)∧0<LENGTH U⊃TRUE is replaced by:
TRUE
```

(iii) Now EKL starts the induction step. It espands the definitions of uniqueness. length and uses the fact *Successorless* (which is in s impinf o).

```
;the term UNIQUENESS(X.U) is replaced by:
¬MEMBER(X,U)∧UNIQUENESS(U)
;the term N'<LENGTH (X.U) is replaced by:
N<LENGTH U
;the term (¬MEMBER(X,U)∧UNIQUENESS(U))∧N<LENGTH U is replaced by:
¬MEMBER(X,U)∧UNIQUENESS(U)∧N<LENGTH U
```

(iv) In expanding fstposition, EKL finds two nested LISP conditionals:

```
;the term FSTPOSITION(X.U,NTH(X.U,N')) is replaced by:
IF ¬MEMBER(NTH(X.U,N'),X.U) THEN NIL ELSE (IF X=NTH(X.U,N') THEN 0 ELSE
FSTPOSITION(U,NTH(X.U,N'))')
;the term MEMBER(NTH(X.U,N'),X.U) is replaced by:
NTH(X.U,N')=X∨MEMBER(NTH(X.U,N'),U)
;the term NTH(X.U,N') is replaced by:
NTH(U,N)
;the term NTH(U,N)=X is replaced by:
FALSE
```

Line 3 has been used here.

```
;the term NTH(X.U,N') is replaced by:
NTH(U,N)
;the term MEMBER(NTH(U,N),U) is replaced by:
TRUE
```

Here EKL has used the fact *Nthmember*.

```
;the term FALSE∨TRUE is replaced by:
TRUE
;the term ¬TRUE is replaced by:
FALSE
```

The if clause of the outermost conditional is therefore false (see the first line after *(iv)*). Now EKL moves to the else clause and finds the innermost conditional.

```
;the term NTH(X.U,N') is replaced by:
NTH(U,N)
;the term X=NTH(U,N) is replaced by:
FALSE
```

Line 3 has been used here again to see that the if clause of the innermost conditional is false. Hence EKL considers the else clause, i.e.

```
FSTPOSITION(U,NTH(X.U,N'))'
```

(see the first line after *(iv)*).

```
;the term NTH(X.U,N') is replaced by:
NTH(U,N)
;the term FSTPOSITION(U,NTH(U,N)) is replaced by:
N
```

Here the induction hypothesis has been used.

```
;the term IF FALSE THEN 0 ELSE N' is replaced by:
N'
;the term IF FALSE THEN NIL ELSE N' is replaced by:
N'
```

This concludes the evaluation of the term FSTPOSITION(X.U,NTH(X.U,N')). The result follows by standard rewriting.

```
;the term N'=N' is replaced by:
TRUE
;the term ¬MEMBER(X,U)∧UNIQUENESS(U)∧N<LENGTH U⊃TRUE is replaced by:
TRUE
;the term (UNIQUENESS(U)∧N<LENGTH U⊃FSTPOSITION(U,NTH(U,N))=N)⊃TRUE is
replaced by:
TRUE
;the term TRUE∧TRUE∧TRUE is replaced by:
TRUE
;the term ∀U N X.TRUE is replaced by:
TRUE
;the term TRUE⊃(∀U N.UNIQUENESS(U)∧N<LENGTH U⊃FSTPOSITION(U,NTH(U,N))=N)
is replaced by:
∀U N.UNIQUENESS(U)∧N<LENGTH U⊃FSTPOSITION(U,NTH(U,N))=N
;∀U N.UNIQUENESS(U)∧N<LENGTH U⊃FSTPOSITION(U,NTH(U,N))=N
```

El

2.10. Injectivity and Uniqueness.

We already pointed out that, in order to represent the property 'each member of a. list u occurs just once in the list u', we can use either the recursively defined predicate uniqueness

```
∀u x.uniqueness nil ∧
     (uniqueness(x.u)≡¬member(x,u)∧uniqueness(u)),
```

or the predicate inj, defined using a bounded quantifier.

```
;injectivity
;another predicate for uniqueness

(proof inj)

(decl (inj) (type: |ground→truthval|))
(define inj
        |∀u.inj(u)=∀n m.n<length(u)∧m<length(u)∧nth(u,n)=nth(u,m)⊃n=m|)
(label injdef)
```

The proof of equivalence of the two predicates can be found in the Appendix.

```
∀U.UNIQUENESS(U)≡INJ(U)
(label uniqueness,injectivity)
```

Clearly the predicate uniqueness is more convenient, in a. proof by induction on lists. An example is the previous Lemma *Fstposition Nth*: a. direct proof of

```
∀U N.INJ(U)∧N<LENGTH U⊃FSTPOSITION(U,NTH(U,N))=N
```

would be much longer.

2.11. The notions of Finite Union and Finite Sum.

We introduce functions that perform finite sums and finite unions, i.e. given f: N→ N. the operation

$$\sum_{m<n} f(m)$$

and given F: **N**→ A, where A is a. collection of sets, the operation

$$\bigcup_{m<n} F(m).$$

The recursively defined predicates `all` and some can be used instead of the bounded quantifiers "for all $m < n$, **a(** m)" and "for some $m < n$, a(m). The proof of *Pigeonfact* shows an effective use of all.

```
(proof sums>

1.  (decl allnum (type: |ground⊗@set→truthval|)
          (syntype: constant))
2.  (decl somenum (type: |ground⊗@set→truthval|)
          (syntype: constant))
3.  (decl (numseq f) (type:|ground→ground|))
4.  (decl sum (type: |(@numseq)⊗(@n)→(@n) I ) (syntype: constant))
5.  (decl setseq (type: |@n→@set|))
6.  (decl un (type: |(@setseq)⊗(@n)→(@set) I) (syntype: constant))

    ; axiom for allnum
7.  (defax allnum |∀n a.allnum(0,a)∧(allnum(n',a)≡a(n)∧allnum(n,a)) |)
    (label allnumdef )

    ;axiom for somenum
8.  (defax somenum |∀n a.¬somenum(0,a)∧(somenum(n',a)≡a(n)∨somenum(n,a))|)
    (label somenumdef)

9.  (defax sum
      |∀n numseq.sum(numseq,0)=0∧
                 sum(numseq,n')=sum(numseq,n)+numseq(n)|)
    (label sumdef )

10. (defax un
      |∀n setseq.un(setseq,0)=emptyset∧
                 un(setseq,n')=un(setseq,n)∪setseq(n)|)
    (label undef )
```

Finally we have a. recursive predicate to identify finite sequences of disjoint sets.

```
9.  (decl disj,pair (type: |(@set⊗@set)→truthval|))
10. (define disj,pair |∀a b.disj_pair(a,b)=emptyp(a∩b)|)
    (label disj,pair,def)

11. (decl disjoint (type: |((ground→@set)⊗ground)→truthval|))
12. (defax disjoint
      |∀n setseq.
         disjoint(setseq,0)∧
         disjoint(setseq,n')=(disjoint(setseq,n)∧
                                  disj_pair(un(setseq,n),setseq(n)))|)
    (label disjoint,def)
```

The following line gives the condition for sum to be defined:

```
    ;sumsort

3.  (ue (a |λn.allnum(n,λm.natnum numseq(m))⊃natnum sum(numseq,n)|)
        proof-by-induction (open allnum sum))
    ;∀N.ALLNUM(N,λM.NATNUM(NUMSEQ(M)))⊃NATNUM(SUM(NUMSEQ,N))

4.  (rw * (use allnumfact mode: exact direction: reverse))
    ;∀N.(∀M.M<N⊃NATNUM(NUMSEQ(M)))⊃NATNUM(SUM(NUMSEQ,N))
    (label sumsort) ■
```

2.12. The not ion of Multiplicity.

The function mult counts the number of members in a list u that satisfy the predicate a.

```
    (proof multiplicity)

1.  (decl mult (type: |(ground⊗@set)→ground|))
2.  (defax mult |∀x u a.mult(nil,a)=0∧
                  mult(x.u,a)=if a(x) then mult(u,a)' else mult(u,a)|)
    (label mult,def)
```

The following fact about multiplicity is easy to prove.

```
3.  (ue (phi |λu.∀a.natnum(mult(u,a))|) listinduction
        (use mult,def mode: always))
    (label simpinfo) (label multfact)     ■
```

Lemma *2.8. (Length Mult)*

VU A.MULT(U,A)≤LENGTH U

Proof. There are two cases in the inductive step. If x does not satisfies a then

mult(x.u,a)=mult(u,a)≤length(x.u)

follows from the definitions and the induction hypothesis.

Otherwise

```
mult(x.u,a)≤length(x.u)
```

follows from the definitions and the induction hypothesis (using SUCCESSORLESSEQ, which is in simpinfo).

```
;multiplicity is lesseq length

;labels:LESSEQ,LESSEQ,SUCC
;∀N M.N≤M⊃N≤M'

;labels: SIMPINFO SUCCESSORFACTS SUCCESSORLESSEQ
;∀N M.N'≤M'≡N≤M

4.  (ue (phi |λu.mult(u,a)≤length(u)|) listinduction
        (open mult length) (use lesseq_lesseq_succ)
        (part 1#1 (open lesseq)))
    ;∀U.MULT(U,A)≤LENGTH U
    (label length_mult) ■
```

Lemma 2.9. *(Member Mult)*

```
∀U Y A.MEMBER(Y,U)∧A(Y)⊃1≤MULT(U,A)

;if there is a member, multiplicity is not zero

5.  (ue (phi |λu.∀y a.member(y,u)∧a(y)⊃0<mult(u,a)|) listinduction
        (open mult member) (use normal mode: always))
    ;∀U Y A.MEMBER(Y,U)∧A(Y)⊃0<MULT(U,A)

6.  (rw * use less-lesseqsucc mode: always))
    ;∀U Y A.MEMBER(Y,U)∧A(Y)⊃1≤MULT(U,A)
    (label member_mult) ■
```

Lemma 2.10. *(Mult Nthcdr)*

```
∀N A U.N<LENGTH U⊃MULT(NTHCDR(U,N),A)≤MULT(U,A)
```

Mult Nthcde is only slightly more difficult. Line 8 is needed to help the rewriter in line 9. The problem in line 9 is the following: we want to expand the definition of mult in the following argument for the induction step: if `a(nth(u,n))`, then

```
mult(nthcdr(u,n),a) = mult(nth(u,n).nthcdr(u,n'),a) = mult(nthcdr(u,n'),a)'
```

otherwise

```
mult(nthcdr(u,n),a) = mult(nth(u,n).nthcdr(u,n'),a) = mult(nthcdr(u,n'),a)
```

But

```
mult(nthcdr(u,n'),a)' ≤ mult(u,a)
```

implies, using the fact *Succ Lesseq Lesseq*,

$$\texttt{mult(nthcdr(u,n'),a)} \leq \texttt{mult(u,a)}$$

Therefore, in both cases

$$\texttt{mult(nthcdr(u,n),a)} \leq \texttt{mult(u,a)}$$

implies'

$$\texttt{mult(nthcdr(u,n'),a)} \leq \texttt{mult(u,a)}.$$

This involves a. combination of rewriting and logical reasoning: the definition of mult is expanded into a *if . . . then ... else* form and the instance of *Succ Lesseq Lesseq* is an implication. We help EKL by giving the logical step described above as a separate rewriter (line 8) using *Trans Cond*.

```
;labels:TRANS,COND
;∀P Q R.(Q⊃R)∧(IF P THEN Q ELSE R)⊃R

;labels:SUCC,LESSEQ,LESSEQ
;∀M N.M'≤N⊃M≤N

8.   (ue ((q.|mult(nthcdr(u,n'),a)'≤mult(u,a)|)
          (r.|mult(nthcdr(u,n'),a)≤mult(u,a)|)
          (p.|a(nth(u,n))|))
        trans,cond
        (use succ_lesseq_lesseq
             ue: ((m.|mult(nthcdr(u,n'),a)|)
                  (n.|mult(u,a)|)) mode: exact ))
     ;(IF A(NTH(U,N)) THEN MULT(NTHCDR(U,N'),A)'≤MULT(U,A)
     ;    ELSE MULT(NTHCDR(U,N'),A)≤MULT(U,A))⊃
     ;MULT(NTHCDR(U,N'),A)≤MULT(U,A)

     ;conclusion

9.   (ue (a |λn.∀a u.n<length(u)⊃mult(nthcdr(u,n),a)≤mult(u,a)|)
        proof-by-induction
        (part 1#1 (open lesseq)) succ,less-less
        (part 1#2#1#1 (use nthcdr,car,cdr mode: always))
        (open mult) * )
     ;∀N A U.N<LENGTH U⊃MULT(NTHCDR(U,N),A)≤MULT(U,A)
     (label mult_nthcdr)   ■


     ;mult emptyset

     (ue (phi |λu.mult(u,emptyset)=0|) listinduction
         (part 1 (open emptyset mult)))
     ;∀U.MULT(U,EMPTYSET)=0
     (label simpinfo) (label emptyfacts)     ■
```

2.12.1. Multiplicity Implies Injectivity.

The following Lemma. embodies the main use of the notion of *multiplicity*. If the number of the occurrences of every member of a list v is 1, then the list has the *injectivity* property: for i, j < length(v).

$$\mathtt{nth(v,i)=nth(v,j)\supset i=j}.$$

We use the following fact: if nth(v, i) = nth(v,j and i<j, then the *multiplicity* of the set mkset nth(v,i) is at least 2.

```
∀V I J.I<J∧J<LENGTH V∧NTH(V,I)=NTH(V,J)⊃
        2≤MULT(V,MKSET(NTH(V,I)))
(label  multinj-computation)
```

The proof of *Multinj Computation,* a. consequence of the lemmata, *Nth in Nthcdr* and *Member Mult*, is left to the Appendix.

Lemma 2.11. (*Mult Inj*)

```
∀V.(∀K.K<LENGTH V⊃MULT(V,MKSET(NTH(V,K)))=1)⊃INJ(V)
```

Proof. At lines 3 and 3 we instantiate *Multinj Computation* and we use line 1 to derive that if $i < j$ or $j < i$, then $2 \leq 1$. Now we esploit semantic attachment: EKL knows that $2 < 1$ and $2 = 1$ are false. An application of the trichotomy concludes the proof.

```
1.  (assume |∀k.k<length v⊃mult(v,mkset(nth(v,k)))=1|)
    (label mil)

2.  (assume |i<length v∧j<length v∧nth(v,i)=nth(v,j)|)
    (label mi2)

3.  (ue ((v.v)(i.i)(j.j)) multinj-computation mi2
        (use mil ue: ((k.i)) mode: exact) (open lesseq))
    ;¬I<J
    ;deps: (MI1 M12)

4.  (ue ((v.v)(i.j)(j.i)) multinj,computation mi2
        (use mil ue: ((k.j)) mode: exact) (open lesseq))
    ;¬J<I
    ;deps: (MI1 MI2)

5.  (derive |i=j| (trichotomy * -2))
    ;deps: (MI1 MI2)

6.  (ci mi2)
    ;I<LENGTH V∧J<LENGTH V∧NTH(V,I)=NTH(V,J)⊃I=J
    ;deps: (MI1)

7.  (trw linj v| (open inj * )
    ;INJ(V)
    ;deps: (MI1)
```

```
8.  (ci mil)
    ;(∀K.K<LENGTH V⊃MULT(V,MKSET(NTH(V,K)))=1)⊃INJ(V)
    (label mult_inj) ■
```

2.12.2. The Multiplicity of a Disjoint Union is the Sum of Multiplicities.

Consider a list and two sets (say, the sets of occurrences of two different S-espressions in the list). If the sets are disjoint, then the sum of multiplicities is the multiplicity of the union (Lemma. *Multsum*). Lemma *Multsum* generalizes to any finite sequence of disjoint sets (Lemma. *Mult of Un is Sum Mult*).

Lemma 2.12. (*Multsum*)

```
∀U.DISJ_PAIR(A,B)⊃MULT(U,A∪B)=MULT(U,A)+MULT(U,B)
```

Proof: By induction on u. For u = NIL, all values of mult are 0. Assume the result for u. The assumption that a and b are a disjoint pair of sets means that the intersection of a and b is empty. If not $x \in a$ and not $x \in b$, then induction hypothesis gives the result. If either $x \in a$ or $x \in b$, then

$$\texttt{mult(x.u,a}\cup\texttt{b)} = \texttt{mult(u,a}\cup\texttt{b)'} = \texttt{(mult(u,a)+mult(u,b))'} = \texttt{mult(x.u,a)+mult(x.u,b)}$$

– the induction hypothesis is used to establish the second equality.

The mechanical proof is one line long:

```
    (proof multsum)
1.  (ue (phi |λu. disj_pair(a,b)⊃mult(u,a∪b)=mult(u,a)+mult(u,b) I)
        listinduction
        (part 1 (open mult union disj,pair emptyp intersection)
                (use normal mode: always))
        (part 1 (der)) )
    (label multsum)   ■
```

The lemma, *Multsum* is used in the induction step in the proof of the next fact:

Lemma *2.13.* (*Mult of Un is Sum Mult*) *If all the sets of the sequence* setseq *are pairwise disjoint, then*

$$\text{mult}(u, \bigcup_{m<n} \text{setseqjm})) = \sum_{m<n} \texttt{mult}(u, \texttt{setseq}(m)) :$$

```
VSETSEQUN.DISJOINT(SETSEQ,N)⊃
          MULT(U,UN(SETSEQ,N))=SUM(λX1.MULT(U,SETSEQ(X1)),N)
```

Proof. By induction on *n*. For *n = 0,*

$$\bigcup_{m<0} \texttt{setseq}(m)$$

is the empty set, whose *multiplicity* is 0 (by simpinfo). and

$$\sum_{m<0} \texttt{mult}(u, \text{setseq}(m))$$

is 0 too.

Assume the result for n. Now

$$\texttt{disjoint(setseq,}n'\texttt{)}$$

implies

$$\texttt{disjpair(un(setseq,}n\texttt{),setseq(}n\texttt{));}$$

this implies, using MULTSUM

$$\text{mult}\Big(u, \bigcup_{m<n'} \texttt{setseq}(m)\Big) = \text{mult}\ (u, \bigcup_{m<n} \text{setseq}(n2)) + \texttt{mult}(u, \texttt{setseq}(n)),$$

which is, by definition of un and induction hypothesis

$$= \sum_{m<n} \texttt{mult(u,setseq(}m\texttt{))} + \texttt{mult(u,setseq(}n\texttt{))} = \sum_{m<n+1} \texttt{mult(u,setseq(}m\texttt{))}.$$

Here the mechanical proof is again one line long!

```
(proof mult_of_un_is_sum_mult)

1.  (ue ( a |λn.disjoint(setseq,n)⊃
            mult(u,un(setseq,n))=sum(λx1.mult(u,setseq(x1)),n)|)
      proof-by-induction
      (open disjoint un sum mult ) multfact
      (use multsum mode: exact) (use normal mode: always))
    ;∀N.DISJOINT(SETSEQ,N)⊃
    ;MULT(U,UN(SETSEQ,N))=SUM(λX1.MULT(U,SETSEQ(X1)),N)
    (label mult,of_un_is_sum_mult)    ■
```

3. Notions of Application.

We give the basic facts about application, injection and permutation using two representations for finite functions: functions as association lists and functions as lists of numbers.

3.1. *Function Application using Association Lists.

Our first. approach uses association lists. We recall the recursive definition of **alist** (see the Appendix) and present the main definitions (see also the Introduction 1.5.2).

```
55. (decl (alist) (type: ground) (sort: alistp))
56. (axiom |∀alist. listp alist|)
    (label simpinf o)

57. (axiom |∀u.alistp u ≡ (¬null u ⊃
   ¬atom car u∧atom car (car u)∧alistp(cdr u))|)
    (label alistdef 1)

58. (axiom |∀xa y alist .alistp nil A alistp (xa.y) .alist I)
    (label alistdef) (label simpinfo)

    (wipe-out)
    (get-proofs nth)

    (proof appalist)
1.  (decl dom (type: |GROUND→GROUND|))
2.  (defax dom |∀xa y alist .dom nil=nil∧
                        dom((xa.y).alist)=xa.dom alist|)
    (label domdef)

3.  (decl range (type: |GROUND→GROUND|))
4.  (defax range |∀xa y alist.range nil=nil∧
                        range((xa.y).alist)=y.range alistl)
    (label rangedef )

5.  (decl functp (type: |GROUND→TRUTHVAL|))
6.  (define functp |∀alist .functp(alist)≡uniqueness dom(alist) | )
    (label functdef)

7.  (decl injectp (type: |GROUND→TRUTHVAL|))
8.  (define injectp
       |∀alist.injectp(alist)≡functp(alist)∧uniqueness range(alist)|)
    (label injectdef)

9.  (decl (appalist) (type: |ground⊗ground→ground|))
10. (define appalist |∀alist y.appalist(y,alist)=cdr assoc(y,alist)|)
    (label appalistdef)
```

Let $\mathbf{alist}_f$ represent the function f. As noticed above, $\mathtt{dom}(\mathbf{alist}_f)$ and $\mathtt{range}(\mathbf{alist}_f)$ do not give the domain and the range of f: rather they *list* the domain and the range of the function in the ordering given by the association list $\mathbf{alist}_f$. To abstract from such ordering we use the functional mklset. (Given a list u, mklset (u) is the set of members of u-identified. as usual. with the predicate λx.member(x,u)).

As we pointed out in the introduction, the same function can be represented by several association lists. in fact by the equivalence class of association lists. The predicate samemap is the appropriate equivalence relation: two association lists alist 1 and alist2 represent the same map if

(i) they are 'defined' on the same set. i.e. their domains are the same *as sets*, and

(ii) for all y, appalist (y ,alistl) = appalist (y ,alist2), i.e. if they 'map' the same elements into the same elements.

Both conditions are needed: appalist (y,alist) may be NIL either because the pair (y.NIL) belongs to alist or because y does not. belong to dom(alist); we do not want to identify the two cases.

```
11. (decl (samemap) (type: |ground⊗ground→truthval|))
12. (define samemap
    (define samemap
       |∀alist alist1.samemap(alist,alist1)≡
                     mklset dom(alist)=mklset dom(alist1)∧
                     (∀y.y∈mklset dom(alist)⊃
                          appalist(y,alist)=appalist(y,alist1)) I)
    (label samemapdef )

13. (define permutp |∀alist.permutp(alist)≡
             functp(alist)∧mklset(dom(alist))=mklset(range(alist))|)
    (label permutp,def)

14. (axiom |∀chi.chi(nil)∧(∀xa y alist.chi(alist)⊃chi((xa.y).alist))⊃
               (∀alist.chi(alist))|)
    (label alistinduction)
```

Alist Induction is easily derivable from *Listinduction* (see the Appendix).
The following facts are very easy to prove:

```
    (proof alistfacts)

    ; domsort

1.  (ue (chi |λalist.listp dom(alist) |) alistinduction (open dom))
    ;VALIST.LISTP DOM(ALIST)
    (label domsort) (label simpinfo)

    ; rangesort

2.  (ue (chi |λalist.listp range(alist)|) alistinduction (open range))
    ;∀ALIST.LISTP RANGE(ALIST)
    (label rangesort) (label simpinfo)

    ;domlength

3.  (ue (chi |λalist.length dom alist=length alist|) alistinduction
        (open dom) )
```

```
;VALIST.LENGTH (DOM(ALIST))=LENGTH ALIST
(label domlength)

;domrangelength

4. (ue (chi |λalist.length(dom alist)=length(range alist)|)
       alistinduction
       (open dom range))
   ;∀ALIST.LENGTH (DOM(ALIST))=LENGTH (RANGE(ALIST))
   (label domrangelength)

   ;appalistsort

5. (ue (chi |λALIST.SEXP APPALIST(Y,ALIST)|)
       alistinduction
       (part 1 (open appalist assoc)))
   ;∀ALIST.SEXP APPALIST(Y,ALIST)
   (label appalistsort)(label simpinfo)

   ;trivial appalist

6. (ue (chi |λalist.¬(y∈mklset dom(alist))⊃appalist(y,alist)=nil|)
       alistinduction
       (part 1 (open epsilon mklset dom appalist assoc member)))
   ;∀ALIST.¬Y∈MKLSET(DOM(ALIST))⊃APPALIST(Y,ALIST)=NIL
   (label trivial,appalist)
```

samemap is an equivalence relation:

```
7. (trw |samemap(alist,alist)|(open samemap))
   ;SAMEMAP(ALIST,ALIST)
   (label samemap_equivalence)

8. (trw |samemap(alist,alist1)⊃samemap(alist1,alist)|
        (open samemap mklset dom))
   ;SAMEMAP(ALIST,ALIST1)⊃SAMEMAP(ALIST1,ALIST)
   (label samemap,equivalence)

9. (trw |samemap(alist,alist1)∧samemap(alist1,alist2)⊃
         samemap(alist,alist2)|
        (open samemap mklset dom))
   ;SAMEMAP(ALIST,ALIST1)∧SAMEMAP(ALIST1,ALIST2)⊃SAMEMAP(ALIST,ALIST2)
   (label samemap_equivalence)
```

The restriction to elements of the domain in the definition of appalist is not necessary: appalist has a default value, as shown in the line *Trivial Appalist*. The easy proof of equivalence is in the Appendix.

```
10. ∀ALIST1 ALIST2.SAMEMAP(ALIST1,ALIST2)≡
                   (MKLSET(DOM(ALIST1))=MKLSET(DOM(ALIST2))∧
                   (∀X.APPALIST(X,ALIST1)=APPALIST(X,ALIST2)))|)
    (label samemap,defl)
```

3.2. Function Application using Lists of Numbers.

Our second representation of functions uses lists of numbers.

```
    ;definition of application

    (proof appl)

1.  (define appl |∀u i.appl(u,i)=nth(u,i) |)
    (label appldef)
    ;predicates for functions

2.  (decl (into) (type: |ground→truthval|))
3.  (define into
       |∀u.into(u)=(∀n.n<length u⊃natnum nth(u,n)∧nth(u,n)<length u)|)
    (label intodef)

4.  (decl (onto) (type: |ground→truthval|))
5.  (define onto |∀u.onto(u)=(into(u)∧(∀n.n<length u⊃member(n,u)))|)
    (label ontodef)

6.  (decl (perm) (type: |ground→truthval|))
7.  (define perm |∀u.perm(u)=onto(u) |)
    (label permdef)
```

Estensionality is proved using *Doubleinduction*. To do the inductive step, we instantiate twice the assumption of line 3, by replacing i first with 0 (line 4) and then with `i'` (line 5).

```
    (proof extensionality)

    (show doubleinduction)
    ;labels:DOUBLEINDUCTION
    ;∀PHI2.(∀U V X Y.PHI2(NIL,U)∧PHI2(U,NIL)∧(PHI2(U,V)⊃PHI2(X.U,Y.V)))⊃
    ;       (∀U V.PHI2(U,V))

    ;first attempt:
0.  (ue (phi2 |λu v.length u=length v∧
                   (∀i.i<length u⊃nth(u,i)=nth(v,i))⊃u=v|)
        doubleinduction (open nth))
    (∀U V X Y.(LENGTH U=LENGTH V∧
               (∀I.I<LENGTH V⊃NTH(U,I)=NTH(V,I))⊃U=V)⊃
              (LENGTH U=LENGTH V∧
               (∀I.I<LENGTH V'⊃NTH(X.U,I)=NTH(Y.V,I))⊃X.U=Y.V))⊃
    (∀U V.LENGTH U=LENGTH V∧(∀I.I<LENGTH V⊃NTH(U,I)=NTH(V,I))⊃U=V)


1.  (assume |LENGTH U=LENGTH V∧(∀I.I<LENGTH V⊃NTH(U,I)=NTH(V,I))⊃U=V|)
    (label ext1)

2.  (assume |LENGTH U=LENGTH V|)
    (label ext2)

3.  (assume |∀I.I<LENGTH V'⊃NTH(X.U,I)=NTH(Y.V,I)|)
```

```
     (label ext3)

4.   (ue (i 0) * ext2)
     ;X=Y
     (label ext4)
     ;deps: (EXT2 EXT3)

5..  (ue (i |i'|) ext3 ext2)
     ;I<LENGTH V⊃NTH(U,I)=NTH(V,I)
     (label ext5)
     ;deps: (EXT2 EXT3)

6.   (derive |u=v| (extl ext2 ext5))
     (label ext6)
     ;deps: (EXT1 EXT2 EXT3)

7.   (trw |x.u=y.v| (use ext4 ext6 mode: exact))
     ;X.U=Y.V
     ;deps: (EXT1 EXT2 EXT3)

8.   (ci (ext2 ext3))
     ;LENGTHU=LENGTHV∧(∀I.I<LENGTHU'⊃NTH(X.U,I)=NTH(Y.V,I))⊃X.U=Y.V
     ;deps: (EXT1)

9.   (ci extl)
     ;(LENGTHU=LENGTHV∧(∀I.I<LENGTHU⊃NTH(U,I)=NTH(V,I))⊃U=V)⊃
     ;(LENGTHU=LENGTHV∧(∀I.I<LENGTHU'⊃NTH(X.U,I)=NTH(Y.V,I))⊃X.U=Y.V)

10. (ue (phi2 |λu v.length u=length v∧
                  (∀i.i<length u⊃nth(u,i)=nth(v,i))⊃u=v|)
        doubleinduction (open nth) * )
    ;∀UV.LENGTHU=LENGTHV∧(∀I.I<LENGTHV⊃NTH(U,I)=NTH(V,I))⊃U=V
    (label extensionality) ■

11. (trw |∀u i.i<length u ⊃ sexp(appl(u,i))∧member(appl(u,i),u)|
         (open appl) nthmember)
    ;∀U I.I<LENGTH U⊃SEXP APPL(U,I)∧MEMBER(APPL(U,I),U)
    (label simpinfo) (label applfacts)      ■
```

2.13. **Conclusion of Part I.**

It may be appropriate to conclude the first part by some remarks and guidelines for the heuristics of particular proofs of EKL. In the second part we will make some suggestions how to choose among mathematical representations and linguistic variants, how to organize the proofs. how to break them into lemmata and how to improve the efficiency of proofs.

How should a user proceed'?

1. *First, we must make sure that we understand the mathematical notions and have a proof strategy that works on paper.* In particular:

—*If a proof* by *induction is needed,* EKL *will not give hints on the form of induction.*

-*Even if the result follows by expanding the definitions nnd making appropriate substitu tions. we cannot expect the rewriting process to find the right substitutions by itself.*

As a proof checker, EKL is not designed to cope with the danger of combinatorial explosion. EKL commands give the user many ways to control and direct the rewriting process according to her (his) proof strategy.

2. Two methods are available in searching for a proof.

—*Search by trial and error.* Try to obtain a proof in a single line. If this does not succeed. use the output of EKL to establish what other information is needed and try again.

—*Expand the proof, using explicitly the logic decision procedure in the style of Natural Deduction.* This is safer, but time consuming.

3. Suppose that, according to the first alternative, we ask EKL to rewrite a certain formula A to true (or a certain term t to t') and EKL gives instead some error message or returns $A \equiv B$ (or $t = u$). The output of EKL and the form of B (or of u) always give useful information. Rewriting may fail because

(i) *Type conditions are not satisfyed.* In this case EKL will return an error message. There may be a. parsing error. Or we need to modify some definition. Otherwise, we tried to prove something that cannot be expressed by EKL.

```
(proof foo)

(trw |p⊃p⊃p|)
;P is unknown.
;type-check: 3
;does not apply in P⊃P⊃P
:-it currently has type (TRUTHVAL⊗TRUTHVAL)→TRUTHVAL
i.

(trw |f(f)|)
;FOO started.
1. ;F is unknown.
; type-check: F
;does not apply in F(F)
;-it currently has type GROUND→GROUND
```

(ii) *Sort conditions are not satisfied.* If something totally obvious didn't. work, this may be the reason. A sorted language is more flexible, but some information is left implicit. To check sorts is of course an essential step: it amounts to check that a term has the intended meaning or that. a function is defined for the given argument.

```
    (proof f 00)
1.  (decl (m n) (type: ground) (sort: natnum))
2.  (decl plus (type: |ground⊗ground→ground|)
          (infixname: |+|) (bindingpower: 930))

3.  (decl (f) (type: |ground→ground|))
4.  (define f |∀n.f(n)=1|)

5.  (decl (g) (type: |ground→ground|))
6.  (define g |∀n.g(n)=n|)

7.  (trw |f (f (n)+1) | (open f))
    ;F(F(N)+1)=1

8.  (trw |f(g(n)+1) | (open f g))
    ;F(G(N)+1)=F(N+1)
```

What's wrong here? Of course we have forgotten the information that n+l is of the sort natnum, so in line 8 the rewriting cannot continue. As soon as this information is available, the rewriting is completed (line 10).

```
9.  (axiom |∀n.natnum(n+1) |)
    (label simpinfo)

10. (trw |f(g(n)+1) | (open f g))
    ;F(G(N)+1)=1
```

One may wonder why the rewriter was successful in line 7. Although we have not defined plus, by semantic attachment 1+1 has its intended meaning and natnum(2) is true.

```
    (setq rewritemessages t)

7.  (trw |f(f (n)+1) | (open f))
    ; the term F(N) is replaced by:
    1
    ;the term 1+1 is replaced by:
    2
    ;the term F(2) is replaced by:
    1
    ;F(F(N)+1)=1
```

(iii) *There are conditions on the rewriting that are not satisfied.* Often the conditions are satisfied, but cannot be verified directly by EKL decision procedure and we need to construct an additional rewriter. (See Example 5.)

(iv) *The use* of *a line is blocked,* because we are rewriting in mode: exact. In this case we may try the same rewriting in mode: always. If the rewriter mode: always causes an infinite loop, then expanding the proof may be our only choice.

(v) *The expression produced by the rewriting is not simpler.* and we are rewriting in the default mode. For instance? at line 12 the line line is not applied when rewriting in default mode. since the expression g(g(n)) resulting from its application would be more complex than f(x). Here it is enough to specify the mode of the rewriting (line 13).

```
11. (assume |f(x)=g(g(n))|)
    (label line)

12. (trw |f(x)| line (open g))
    ;F(X)=F(X)
    ; deps : (LINE)

13. (trw |f(x)| (use line mode: exact)(open g))
    ;F(X)=N
    ;deps: (LINE)
```

(vi) *The line is not applicable, because of a confict of context.*

```
14. (define g |∀x.g(x)=f(x) |)
    (label gdef)

15. (trw |g(x) | (open g) line)
    ;context of line  GDEF cannot be adjoined: the atom G in line GDEF has
    two definitions: one from line 6 and the other from GDEF
```

4. The following is a nontrivial esample, in which there is additional logical structure to be considered in rewriting. We consider the proof of Lemma 2.10 *Mult Nthcdr.* Section 2.12. Here the rewriting line is found by interaction with EKL, and from this rather involved espressiou a. general 'propositional schema' is abstracted, to be used in similar contexts.

Example 5.

```
∀N A U.N<LENGTH U⊃MULT(NTHCDR(U,N),A)≤MULT(U,A)
```

This is a statement about the sublists of a given list u formulated in terms of the function nthcdr. It may be convenient to use a proof by induction on *n.* In the base case, since nthcdr (u ,0) is u, EKL has to know only what nthcdr and ≤ mean. It is enough, therefore, to say (open lesseq) in the part of the induction axiom corresponding to the base case (the definition of nthcdr is in simpinfo). To do the induction step one can formalize the informal argument given in the test. assuming

```
n<length u⊃mult(nthcdr(u,n),a)≤mult(u,a)
```

and deriving

```
n'<length u⊃mult(nthcdr(u,n'),a)≤mult(u,a)
```

To avoid an explicit proof, we notice that we can easily induce EKL to rewrite the inductive step as

```
(N<LENGTH U⊃MULT(NTH(U,N).NTHCDR(U,N'),A)≤MULT(U,A))⊃
(N-LENGTHU⊃MULT(NTHCDR(U,N'),A)≤MULT(U,A))
```

Then `mult` is expanded and the conditional clause is pushed out; hence the same line rewrites to

```
(N<LENGTH U⊃
   (IF A(NTH(U,N))
       THEN MULT(NTHCDR(U,N'),A)'≤MULT(U,A)
       ELSE MULT(NTHCDR(U,N'),A)≤MULT(U,A)))⊃
(N-LENGTH U⊃MULT(NTHCDR(U,N'),A)≤MULT(U,A)) .
```

This is not very perspicuous. The key point is to realize that the structure of the logical argument can be summarized in the formula *Trans Cond*:

```
∀P Q R.(Q⊃R ∧(IF P THEN Q ELSE R)⊃R,
```

where Q is

```
mult(nthcdr(u,n'),a)'≤mult(u,a),
```

where R is

```
mult(nthcdr(u,n'),a)≤mult(u,a),
```

and P is

```
a(nth(u,n)).
```

Clearly `Q⊃R` follows from elementary arithmetic (fact *Succ Lesseq Lesseq)*. To do the inductive argument in one step we need only prepare one rewriter (line 8 in the test.):

```
(IF A(NTH(U,N))
    THEN MULT(NTHCDR(U,N'),A)'≤MULT(U,A)
    ELSE MULT(NTHCDR(U,N'),A)≤MULT(U,A))⊃
MULT(NTHCDR(U,N'),A)≤MULT(U,A)
```

and use the following fact of elementary arithmetic (*Succ Less Less):*

```
N'<LENGTH U⊃N<LENGTH U
```

The simplification of this proof is certainly worth the effort. Indeed the argument used here is quite common in proofs about recursively defined objects. There is a good chance that the rewriter *Trans Cond* may be applied in similar cases. □

5. Finally it may be the case that, despite our attempts, we cannot find by trial and error the appropriate rewriter. Then we espand our proof in a 'Natural Deduction style': e.g. in a proof by induction we try to prove the base case, we assume the induction hypothesis and try to prove the conclusion of the inductive step. If the latter is in turn an implication. we assume the antecedent etc. In the process, we may expand definitions, perform substitutions, etc. Moreover, we may need to prove other lemmata also by induction. The process is not easily described in general terms. since *there is no general analysis of higher order inductive* proofs *in Natural Deduction.* as we remarked earlier. In practice it. is quite clear what to do, although several options may be open, especially when we are engaged in a proof by contradiction.

6. Once the derivation is found we may *try to collapse it in few steps.* For example. it may may be clear which formulas could be taken as rewriters.

In trying to replace logical deduction by rewriting, we find some steps harder to handle than others.

(i) A line resulting from the cases command, i.e. the conclusion of a proof by cases. corresponds to rewriting a disjunction in the antecedent of an implication. This can be handled by using the rewriter NORMAL as explained in Example 1.

(ii) Argument by contradiction and steps involving negation may require some help. For instance. although EKL can easily derive $\neg B \supset \neg A$ from $A \supset B$, it may not do this step in the contest of conditional rewriting.

(iii) Quantifiers may require additional lines, in particular the existential one. If *the result involves a bound quantifier, it may be convenient to replace it by a recursively defined predicate.* For instance

`∀m.m<n⊃A(m),` `∃m.m<n∧A(m),` `∀x.member(x,u)⊃A(x),` `∃x.member(x,u)∧A(x).`

are equivalent to the recursive predicates

`allnum(n,λm.A(m)),` `somenum(n,λm.A(m)),` `allp(λx.A(x),u),` `somep(λx.A(x),u).`

In the contest of inductive proofs, it is convenient to formulate the result by using the recursive predicate and prove this formula first. This method is presented in Examples 6 and 7.

The proof of *Fstposition Nth,* already examined in Example 4, Section 2.9 , is an instance of the process of collapsing a long proof in few lines. First of all, in later applications we use

($\star$) `∀U N.INJ(U)∧N<LENGTH U⊃FSTPOSITION(U,NTH(U,N))=N`

rather than

($\star\star$) `∀U N.UNIQUENESS(U)∧N<LENGTH U⊃FSTPOSITION(U,NTH(U,N))=N`

and we may be tempted to prove directly ($\star$). inj and uniqueness are equivalent predicates. but the latter is a recursively defined predicate, whereas the former has an explicit definition using quantifiers. In the spirit of our suggestion *(iii),* we should try a. proof of ($\star\star$) and indeed we find one four lines long. Moreover, notice that we derive line 3 of the proof from line 2, to allow the use of `¬x=nth(u,n)`, a. negative formula, in rewriting. This is in accordance to our suggestion (*ii).* Looking at the proof. it is completely clear that line 3 will do the job, by considering its effects of the rewriting of line 3. But the form of 3 or the possibility of proving it in two steps may not. have occurred to us at first sight, before a. more detailed proof.

In conclusion, one learns to control the rewriting process of EKL by trial and error: it may be necessary first to write some explicit proofs in order to understand with total clarity the single step of rewriting. Then one may succeed in collapsing the proof into a single step. using a suitable line to reduce logical inferences to steps of rewriting. Several proofs in this paper were obtained in this way and some more may be reduced to a few lines with some additional effort. To make the proofs shorter doesn't mean to make them clearer. As in informal mathematical presentations, a balance has to be found between the necessity of formal precision and the need for clarity. One has to admit, however, that at the present stage it is premature to worry about mechanical proofs being too concise.

PART 2

4. The Pigeon Hole Principle.

In this section we prove the Pigeon Hole principle in second order arithmetic and apply it to show that every finite surjection is an injection, in our two representations.

4.1. The Pigeon Hole Principle in Second Order Arithmetic.

Theorem. (*Pigeonfact*)

```
∀F.(∀N.NATNUM(F(N)))⊃
   (∀N.(∀M.M<N⊃1≤F(M))∧SUM(λK.F(K),N)=N⊃(∀M.M<N⊃1=F(M)))
```

We give two versions of the proof. In the former we prove directly

```
∀F N.(∀M.M<N⊃1≤F(M))∧SUM(λK.F(K),N)≤N⊃(∀M.M<N⊃1=F(M));
```

the presence of quantifiers requires a proof in the style of Natural Deduction.

In the latter we assume ∀N . NATNUM F(N) and we prove

```
∀N.ALLNUM(N,λK.1≤F(K))∧SUM(λK.F(K),N)=N⊃ALLNUM(N,λK.1=F(K));
```

the use of the recusively defined predicate `allnum` allows a. straightforward proof by induction.

First Proof. We need a preliminary fact: if f is defined and has positive values on **0,** . . . . $n-1$. then the function

$$\sum_{m<n} f(m)$$

is strictly increasing. The proof is a. straightforward induction. using in the induction step the lemma *Add Lesseq* (line 7).

```
(wipe-out)
(get-proofs sums)

(proof pigeonfact)

1    (assume |(∀m.m<n⊃natnum f(m)∧1≤f(m))⊃n≤sum(λk.f(k),n)|)
     (label si,indhyp)

2    (assume |∀m.m<n'⊃natnum f(m)∧1≤f(m)|)
     (label si_hyp)

3    (trw |∀m.m<n⊃natnum f (m)∧1≤f(m)|
          (* transitivity-of-order successor1))
     ;∀M.M<N⊃NATNUM(F(M))∧1≤F(M)
     (label sil)

4    (ue ((numseq.|λk.f(k)|)(n.n)) sumsort * )
     ;NATNUM(SUM(λK.F(K),N))
     (label sisort)
```

```
5.  (derive |n≤sum(λk.f(k),n)|(si1 si,indhyp))
    (label si2)

6.  (ue (m n) si,hyp successor1)
    ;NATNUM(F(N))∧1≤F(N)
    (label si3)
    ;deps: (SI,HYP)
```

We need *Add Lesseq*:

```
    ;labels:ADD,LESSEQ
    ;∀N M.N≤M∧1≤K⊃N'≤M+K

7.  (ue ((n.n)(k.|f(n)|)(m.|sum(λk.f(k),n)|))
        add,lesseq (sisort si2 si3))
    ;N'≤SUM(λK.F(K),N)+F(N)
    ;deps: (SI_INDHYP SI_HYP)

8.  (ci si_hyp)
    ;(∀M.M<N'⊃NATNUM(F(M))∧1≤F(M))⊃N'≤SUM(λK.F(K),N)+F(N)
    ;deps: (SI,INDHYP)

9.  (ci si,indhyp)
    ;((∀M.M<N⊃NATNUM(F(M))∧1≤F(M))⊃N≤SUM(λK.F(K),N))⊃
    ;((∀M.M<N'⊃NATNUM(F(M))∧1≤F(M))⊃N'≤SUM(λK.F(K),N)+F(N))

10. (ue (a |λn.(∀m.m<n⊃natnum f(m)∧1≤f(m))⊃n≤sum(λk.f(k),n)|)
        proof-by-induction
        (open sum) zeroleast (use * mode: always))
    ;∀N.(∀M.M<N⊃NATNUM(F(M))∧1≤F(M))⊃N≤SUM(λK.F(K),N)
    (label strictly-increasing)
```

Next we want to show that if the values of f are greater than or equal to 1 and, in addition, the value of

$$\sum_{m=0}^{n-1} f(m)$$

is bounded by n, then the values of f must be equal to 1.

The proof is another simple induction, using in the induction step the lemma. *Add One* (line 19).

```
    ;use
    ;labels:ADD-ONE
    ;(AXIOM |∀K N M.1≤K ∧ N'=M+K ∧ N≤M ⊃ 1=K ∧ N=M|)
```

We will replace f(n) for k and sum(λk .f (k) ,n) for m. Lines 15, 17 and 18 are all the conditions in the antecedent of the Lemma to apply the lemma. At line 18 we use the first part *Strictly Increasing*.

```
;other direction

11. (assume |(∀m.m<n⊃natnum f(m)∧1≤f(m))∧sum(λk.f(k),n)=n⊃(∀m.m<n⊃1=f(m))|)
    (label pf indhyp)

12. (assume |∀m.m<n'⊃natnum f(m)∧1≤f(m)|)
    (label pf _assume)

13. (derive |∀m.m<n⊃natnum f(m)∧1≤f(m)|
              (pf,assume transitivity_of_order successor1))
    (label pf0)

14. (ue ((numseq.|λk.f(k)|)(n.n)) sumsort * )
    ;NATNUM(SUM(λK.F(K),N))
    (label pfsort)
    ;deps : (PF,ASSUME)
```

The following is the first fact needed for the application of the lemma *Add One* We obtain it as an immediate consequence of the assumption of the inductive step.

```
15. (ue (m n) pf,assume successor11
    ;NATNUM(F(N))∧1≤F(N)
    (label pf 1)
    ; deps : (PF,ASSUME)
```

-The second fact,

$$n' = \sum_{m=0}^{n-1} f(m) + f(n),$$

is also an assumption of the inductive step.

```
16. (assume |sum(λk.f(k),n')=n'|)
    (label pf _assume)

17. (rw * (open sum)>
    ;SUM(λK.F(K),N)+F(N)=N'
    (label pf2)
    ;deps: (PF,ASSUME)
```

The third fact,

$$n \leq \sum_{m=0}^{n-1} f(m),$$

is a direct consequence of *Strictly Increasing.*

```
18. (derive |n≤sum(λk.f(k),n)| (strictly-increasing pf0 pfsort))
    (label pf3)
    ;deps: (PF_ASSUME)

19. (ue ((k.|f(n)|)(n.n)(m.|sum(λk.f(k),n)|)) add-one
      (pf1 pf2 pf3 pfsort))
    ;1=F(N)∧N=SUM(λK.F(K),N)
    (label pf4)
    ;deps: (PF_ASSUME)
```

We use the second conjunct to apply the induction hypothesis.

```
20. (derive |∀m.m<n⊃1=f(m)| (pfindhyp pf0 * ))
    (label pf5)
    ;deps: (PF,ASSUME PFINDHYP)

21. (derive |n0=n⊃1=F(n0)| pf4)
    ; deps : (PF,ASSUME)

22. (trw |∀m.m<n'⊃1=f(m)| (use less,succ,lesseq mode: exact)
       (open lesseq) (use normal mode: always) pf5 * )
    ;∀M.M<N'⊃1=F(M)
    ;deps: (PF,ASSUME PFINDHYP)

23. (ci pf,assume)
    ;(∀M.M<N'⊃NATNUM(F(M))∧1≤F(M))∧SUM(λK.F(K),N')=N'⊃(∀M.M<N'⊃1=F(M))
    ; deps : (PFINDHYP)

24. (ci pfindhyp)

25. (ue (a |λn.(∀m.m<n⊃natnum f(m)∧1≤f(m))∧sum(λk.f(k),n)=n⊃
                (∀m.m<n⊃1=f (m))|)
        proof-by-induction * )
    ;∀N.(∀M.M<N⊃NATNUM(F(M))∧1≤F(M))∧SUM(λK.F(K),N)=N⊃(∀M.M<N⊃1=F(M))
```

Check that the result holds for any f:

```
26. (trw |∀f n.(∀m.m<n⊃natnum f(m)∧1≤f(m))∧sum(λk.f(k),n)=n⊃
                (∀m.m<n⊃1=f (m))| * )
    ;∀F N.(∀M.M<N⊃NATNUM(F(M))∧1≤F(M))∧SUM(λK.F(K),N)=N⊃
    ;     (∀M.M<N⊃1=F(M))
    (label pigeonf act)   ■
```

Second Proof. Using the inductive predicate allnum instead of quantifiers, the theorem is proved very quickly.

```
    (wipe-out)
    (get-proofs sums)
    (proof pigeonf act)

1.  (assume |∀n.natnum f(n)|)
    (label sort 1)

2.  (ue ((numseq.|λk.f(k)|)(n.n)) sumsort * )
    ;NATNUM(SUM(λK.F(K),N))
    (label sort2)

3.  (ue (a |λn.allnum(n,λk.1≤f(k))⊃n≤sum(λk.f(k),n)|)
        proof-by-induction
        (open allnum sum) zeroleast (use sort1 sort2 mode: always)
        (use add,lesseq
             ue: ((n.n)(k.|f(n)|)(m.|sum(λk.f(k),n)|)) ))
    (label strictly-increasing)
```

```
;∀N.ALLNUM(N,λK.1≤F(K))⊃N≤SUM(λK.F(K),N)
;deps: (SORT1)

4.  (ue (a |λn.allnum(n,λk.1≤f(k))∧sum(λk.f(k),n)=n⊃
            allnum(n,λk.1=f(k))|)
      proof-by-induction
      (open allnum sum) strictly-increasing sort1 sort2
      (use add-one
          ue: ((k.|f(n)|)(n.n)(m.|sum(λk.f(k) ,n) I)) mode: always))
    ;∀N.ALLNUM(N,λK.1≤F(K))∧SUM(λK.F(K),N)=N⊃ALLNUM(N,λK.1=F(K))

;in more conventional notation:

5.  (rw * (use allnumfact ue: ((a.|λk.1≤f(k)|)(n.n))
            mode: always direction: reverse)
         (use allnumfact ue: ((a.|λk.1=f(k)|)(n.n))
            mode: always direction: reverse))
    ;∀N.(∀M.M<N⊃1≤F(M))∧SUM(λK.F(K),N)=N⊃(∀M.M<N⊃1=F(M))
    ;deps: (SORT1)

6.  (ci sort1)
    ;(∀N.NATNUM(F(N)))⊃
    ;(∀N.(∀M.M<N⊃1≤F(M))∧SUM(λK.F(K),N)=N⊃(∀M.M<N⊃1=F(M)))
    (label pigeonfact) ■
```

Remark. Example 6. Let us consider the heuristics of this theorem. If we formulate *Strictly Increasing* using the inductive predicate 'allnum', and expand the definitions we obtain:

```
0.  (ue (a |λn.allnum(n,λk.natnum f(k)∧1≤f(k))⊃n≤sum(λk.f(k),n)|)
      proof-by-induction (open allnum sum>)
    ;0≤0∧
    ;(∀N.(ALLNUM(N,λK.NATNUM(F(K))∧1≤F(K))⊃N≤SUM(λK.F(K),N))⊃
    ;    (NATNUM(F(N))∧1≤F(N)∧ALLNUM(N,λK.NATNUM(F(K))∧1≤F(K))⊃
    ;     N'≤SUM(λK.F(K),N)+F(N)))⊃
    ;(∀N.ALLNUM(N,λK.NATNUM(F(K))∧1≤F(K))⊃N≤SUM(λK.F(K),N))
```

The main point is to formulate the fact *Add Lesseq*. In other words, we must recognize that the following line would do the job (once we guarantee that f(n) and sum(λk.f(k),n) are natural numbers).

```
0.  (ue ((n.n)(k.|f(n)|)(m.|sum(λk.f(k),n) |)) add-lesseq)
    ;NATNUM(F(N))∧NATNUM(SUM(λK.F(K),N))⊃
    ;(N≤SUM(λK.F(K),N)∧1≤F(N)⊃N'≤SUM(λK.F(K),N)+F(N))
```

Similarly, the theorem is:

```
0.  (ue (a |λn.allnum(n,λk.natnum f(k)∧1≤f(k))∧sum(λk.f(k),n)=n⊃
            allnum(n,λk.1=f(k)) I)
      proof-by-induction (open allnum sum))
    ;(∀N.(ALLNUM(N,λK.NATNUM(F(K))∧1≤F(K))∧
    ,     SUM(λK.F(K),N)=N⊃ALLNUM(N,λK.1=F(K)))⊃
    ,    (NATNUM(F(N))∧1≤F(N)∧ALLNUM(N,λK.NATNUM(F(K))∧1≤F(K))∧
```

```
;     SUM(λK.F(K),N)+F(N)=N'⊃1=F(N)∧ALLNUM(N,λK.1=F(K))))⊃
;(∀N.ALLNUM(N,λK.NATNUM(F(K))∧1≤F(K))∧SUM(λK.F(K),N)=N⊃
;    ALLNUM(N,λK.1=F(K)))
```

Again the key idea. is the fact Add *One*. We need only to construct the following line as a rewriter (and make sure that f(n) and sum(λk.f (k) ,n) are natural numbers).

```
0.  (ue ((k.|f(n)|)(n.n)(m.|sum(λk.f(k),n)|)) add-one)
    ;NATNUM(F(N))∧NATNUM(SUM(λK.F(K),N))⊃
    ;(1≤F(N)∧SUM(λK.F(K),N)+F(N)=N'∧N≤SUM(λK.F(K),N)⊃
    ; 1=F(N)∧N=SUM(λK.F(K),N))
```

A priori. it may seem irrelevant for our capacity of discovering the appropriate steps whether these facts are expressed by bound quantifiers or by recursive predicates. Intuitively, we can say t hat the second representation helps us to 'think recursively' and to focus our attention towards the right inductive step. The mechanization of the proof makes it clear that the second representation is indeed more economica. and gives more precise content to our intuition. □

4.2. Corollary for Application to Lists.

As an application we prove the following corollary. Let. $\{a_1, ... a,\}$ be a. sequence of pairwise disjoint. sets, given by the functional **λm.setseq(m)** and let w be any list of length n.

If the function we consider associates any set a_i with the number of occurrences in w of element s of a_i, then we can certainly define it as a total function and the sum of the values is certainly bound by n, the length of w. In our terminology:

Corollary. (*Pigeonlist*)

```
∀U.DISJOINT(SETSEQ,LENGTH U)⊃
   ((∀M.M<LENGTH U⊃1≤MULT(U,SETSEQ(M)))⊃
   (∀M.M<LENGTH U⊃1=MULT(U,SETSEQ(M))))
```

Proof Consider the function **λk .mult** (w , **setseq(k)**) from N to N.

(1) $$\lambda k.\mathtt{mult}(\mathtt{w},\mathtt{setseq}(k))$$

is a total function. **This** is certainly true, no matter what setseq is by definition of mult (see *Multfact*).

Since the sets setseq(i), for i < length(w), are pairwise disjoint, then

$$\sum_{m<length(w)} \mathtt{mult}(\mathtt{w},\mathtt{setseq}(m)) \leq \bigcup_{i<length(w)} \mathtt{setseq}(i)$$

by the Len-ma 4.7.(*Mult of Un* is *Sum Mult*). Also by the Lemma. 2.8. (*Length Mult*)

$$\bigcup_{i<length(w)} \mathtt{setseq}(i) \leq length(\mathtt{w}).$$

Therefore

$$\sum_{m<length(w)} \texttt{mult(w,setseq(}m\texttt{))} \leq \text{length(w)} \tag{2}$$

Hence we can apply the Theorem *Pigeonfact* to obtain the Corollary.

```
     (proof pigeonlist)

1.   (assume |disjoint(setseq,length w)|)
     (label pl1)

     ;multiplicity less than length

2.   (ue ((u.w)(a.|un(setseq,length w)|)) length_mult)
     ;MULT(U,UN(SETSEQ,LENGTHU))≤LENGTHU
     (label pl2)

3.   (derive |sum(λm.mult(w,setseq(m)),length w)≤length w|
             (mult_of_un_is_sum_mult pl1 pl2))
     (label pl3)

4.   (ue ((f. |λm.mult(u,setseq(m))|)(n.|length u|)) pigeonfact
         pl3 multfact)
     ;(∀M.M<LENGTH U⊃1≤MULT(U,SETSEQ(M)))⊃
     ;(∀M.M<LENGTH U⊃1=MULT(U,SETSEQ(K)))
     ;deps: (PL1)

;the pigeon hole principle on lists

5.   (ci pl1)
     ;DISJOINT(SETSEQ,LENGTHU)⊃
     ;((∀M.M<LENGTH U⊃1≤MULT(U,SETSEQ(M)))⊃
     ; (∀M.M<LENGTH U⊃1=MULT(U,SETSEQ(K))))
     (label pigeonlist) ■
```

4.3. Application of the Pigeon Hole Principle to Lists.

Having proved the Pigon Hole Principle, we will conclude that every map f of a finite set A onto itself is an injection, using our two different representations of finite functions. We could formalize the informal proof. given as a Lemma in the Introduction. Section 1.4. Actually, we could prove a more general result for surjective mappings $f : A \to B$ between finite sets of the same cardinality. (The mechanical proof is described as Example 10 in the Conclusion.) (This approach is described as Example 10 in the Conclusion.)

By restricting ourselves to permutations. we can slightly simplify our proof as follows.

— First, let

$$\{x_1, \ldots, x_n\}$$

be an enumeration without repetition of the set *domain(f)* and let v be the list

$$(y_1 \ldots y_n)$$

where

$$y_j = f(x_i)$$

for some i. v lists *range(f)*, possibly with repetitions. Finally, consider the sequence of sets

$$\{x_1\}, \ldots, \{x_n\}.$$

These sets are disjoint.

— Second, since f is onto A, for each $\{x_i\}$ there is some y_j such that $x_i = y_j$, i.e.

$$|\{j : y_j = x_i\}| \geq 1,$$

or, in our terminology,

$$\texttt{mult(v,mkset(x}_i\texttt{))} \geq 1.$$

Therefore by the Pigeon Hole Principle,

$$|\{j : y_j = x_i\}| = 1.$$

— Finally, since f is into A, each y_i is some x_j. It follows that the set of all sets $\{x_i\}$ is a partition of v, i.e. of *range(f)* (Every element y_j of *range(f)* belongs to one and only one class $[y_j]$.) By the second step, each $[y_j]$ has cardinality 1. It follows easily that if $f(x_i) = f(x_j)$ then $i = j$.

The two representations for finite functions cause some variations in the argument. In both cases, the fact that f is an injection is represented by the fact that each element of S occurs just once in a certain list v. In both cases we use the function mult to count the number of occurrences of elements of a set in a list, i.e. $|\{j : y_j = x_i\}|$.

4.3.1. Application of the Pigeon Hole Principle to Alists.

In this subsection we give the proof that every map of a finite set onto itself is an injection, using the representation of functions by association lists (Theorem *Permutp Injectp*, Section 4.3.5).

If $\texttt{alist}_f$ represents f, the fact that f is a map of a finite set onto itelf is given by the property $\texttt{permutp(alist}_f\texttt{)}$. Then the two lists u = $\texttt{dom(alist}_f\texttt{)}$ and v = $\texttt{range(alist}_f\texttt{)}$ have the same length and contain the same set of elements. The list u has the *uniqueness* property since f is a function. Our ultimate goal is to show that v has the uniqueness property, too.

We search for a partitioning $\{a_i : i \leq n\}$ of v, where n = length(v), namely, a sequence of n nonempty disjoint sets, such that each element of v belongs to some set of the sequence. We know more about u than about v, since u has the *injectivity* property. The idea is to consider the sequence of the sets

$$\{x : x = \texttt{nth(u,}m\texttt{)}\},$$

for $m <$ length(u): in our notation

$$\lambda m.mkset(nth(u,m)):$$

this is a sequence of nonempty sets, whose union is the set of members of u. We can prove that it is *disjoint*, since u is *injective*. Thus it partitions u. It partitions also v, since u and **v** are the same *as sets*.

4.3.2. Step 1: Injectivity implies Disjointness.

Lemma 4.1. (*Inj Disj*)

```
∀U.INJ(U)⊃DISJOINT(λM.MKSET(NTH(U,M)),LENGTH U)
```

Proof. To show this. we prove

```
∀n.inj(u)∧n≤length u⊃disjoint(λm.mkset(nth(u,m)),n)
```

by induction on n (line 13). The theorem follows by taking n = length(u). The essential part of the induction step is proved first as a lemma (line 1'2).

```
    ;inj implies disjoint
    (proof inj,disj)

    ;a main lemma for the induction step

1.  (assume |inj u|)(label injdsj0)

2.  (rw * (open inj))(label injdsjl)
    ;∀N M.N<LENGTH U∧M<LENGTH U∧NTH(U,N)=NTH(U,M)⊃N=M

3.  (assume |n<length u|)(label injdsj2)

4.  (assume |(un(λm.mkset(nth(u,m ),n))(xv)∧(mkset(nth(u,n)))(xv)|)
    (label injdsj3)

    ;need mksetfact

5.  (ue ((u.u)(n.n)) mksetfact (open lesseq) injdsj2)
    ;UN(λM.MKSET(NTH(U,M)),N)=(λX.(∃K.K<N∧NTH(U,K)=X))

6.  (rw injdsj3 (use * mode: exact) (open mkset) injdsj2)
    ;(∃K.K<N∧NTH(U,K)=XV)∧XV=NTH(U,N)
    (label injdsj4)

7.  (define kv |kv<n∧nth(u,kv)=xv| (use *))
    (label injdsj5)

8.  (derive |kv<length u∧nth(u,kv)=nth(u,n)|
            (* injdsj2 transitivity-of-order)
            (use injdsj4 mode: always direction: reverse>>

9.  (derive |kv=n| (injdsj2 * injdsjl))

10. (rw injdsj5 (use * mode: exact) irreflexivity,of-order)
    ;FALSE
    ;deps: (INJDSJO INJDSJ3 INJDSJ2)

11. (ci injdsj3)
    ;¬((UN(λM.MKSET(NTH(U,M)),N))(XV)∧(MKSET(NTH(U,N)))(XV))
```

```
12. (ci (injdsj0 injdsj2))
    ;INJ(U)∧N<LENGTH U⊃
    ;¬((UN(λM.MKSET(NTH(U,M)),N))(XV)∧(MKSET(NTH(U,N)))(XV))
    (label injdsj,lemma)
```

The result follows in two lines.

```
13. (ue (a |λn.inj(u)∧n≤length(u)⊃disjoint(λm.mkset nth(u,m),n) |)
        proof-by-induction
        (open disjoint disj,pair intersection emptyp)
        (use less,lesseqsucc mode: always direction: reverse)
        (use id-lemma mode: always) (part 1#2#1#1 (open lesseq)))
    ;∀N.INJ(U)∧N≤LENGTH U⊃DISJOINT(λM.MKSET(NTH(U,M)),N)

14. (ue (n |length u|) * (open lesseq))
    ;INJ(U)⊃DISJOINT(λM.MKSET(NTH(U,M)),LENGTH U)
    (label inj,disj) ■
```

4.3.3. Step 2: Positive Multiplicity.

Lemma 4.2. (*Permutp Injectp Lemma*)

```
∀U V.MKLSET U=MKLSET V⊃
     (∀M.M<LENGTH U⊃1≤MULT(V,MKSET NTH(U,M)))
```

Proof. We want to show that the multiplicity in v of the set {x : x = nth(u,m)} is positive (line 8) under the assumption that u and v have the same set of elements (line 1). In fact, we obtain a map

$$N_{length(u)} \to N_{length(v)}$$
$$n \mapsto kv,$$

where nth(v,kv) = nth(u,n) (line 6).

```
    (proof permutp,injectp,lemma)

1.  (assume |mklset u=mklset v|)
    (label pil1)

2.  (assume |n<length u|)
    (label pil2)
```

The fact that nth(u,n) $\in \{xv$: member $(x\ v$,u)$\}$ is an immediate consequence of *Nthmember*.

```
3.  (trw |nth(u,n)∈ mklset u| (open epsilon mklset)
  nthmember pil2)
    ;NTH(U,N)∈MKLSET(U)
    ;deps: (PIL2)
```

Here we apply line 1.

```
4.  (rw * (use pil1 mode: exact))
    ;NTH(U,N)∈MKLSET(V)
    ;deps: (PIL1 PIL2)
```

Finally, using *Mklset Fact*, we prove the existence of a kv such that nth(v,kv) = nth(u,n).

```
5.  (rw * (use mklset_fact mode: exact) (open epsilon mkset))
    ;∃K.K<LENGTH V∧NTH(V,K)=NTH(U,N)
    ;deps: (PIL1 PIL2)

6.  (define kv |kv<length(v)∧nth(v,kv)=nth(u,n)| * )
    (label pil3)
    ;deps: (PIL1 PIL2)

7.  (trw |member(nth(v,kv),v)| nthmember pi13)
    ;MEMBER(NTH(V,KV),V)
    (label pil4)
```

Therefore, by the Lemma. *member mult*, the set $\{xv : xv =$ nth(u,n)$\}$ has positive multiplicity in v.

```
a.  (ue ((u.v)(y.|nth(v,kv)|)(a.|mkset nth(u,n)|)) member,mult
(part 1(open mkset)) pi12 pi14 (use pi13 mode: always))
    ;1≤MULT(V,MKSET(NTH(U,N)))
    ;deps: (PIL1 PIL2)

9.  (ci (pill pi12))
    ;MKLSET(U)=MKLSET(V)∧N<LENGTH U⊃1≤MULT(V,MKSET(NTH(U,N)))
```

Cosmetics:

```
10. (derive |∀u v.mklset u=mklset v⊃
                  (∀m.m<length u⊃1≤mult(v,mkset nth(u,m)))| * )
    (label permutp_injectp_lemma) ■
```

4.3.4. Step 3: The Sequence partitions the Range.

Using the result of steps 1 and 2, we will apply the corollary *Pigeonlist* to obtain:

∀m.m<length u⊃1≤mult(v,mkset nth(u,m))

In the final step, for each member x of v we consider the set $\{x\ v : xv = \mathbf{x}\}$ – in our notation we take mkset(nth(v,i)), with x = nth(v,i) for some i – and show that it coincides with some element of the partition constructed in step 2. Hence we can conclude

mult(v,mkset(nth(v,i))) = 1

for all $i <$ length(v). Injectivity of v will follow by the Lemma. *Mult Inj* (Section 2.12.1).

Lemma 4.3. (*Mult Mult*)

```
∀U V.MKLSET(U)=MKLSET(V)∧
     (∀M.M<LENGTH U⊃MULT(V,MKSET(NTH(U,M)))=1)⊃
     (∀I.I<LENGTH V⊃MULT(V,MKSET(NTH(V,I)))=1)
```

Proof. This step is easy. Since u and v are the same *as sets*, the i-th element of v occurs in u. By the inclusion of line 4, we obtain a map

$$\mathbf{N}_{length(v)} \to \mathbf{N}_{length(u)}$$

$$i \mapsto mv$$

where `nth(u,`mv`) = nth(v,`i`)` (line 6). It follows from our main hypothesis (line 2) that the i-th element has just one occurrence in v.

```
(proof mult_mult)

1.  (assume |mklset u = mklset v|)
    (label mm1)

2.  (assume |∀m.m<length u ⊃ mult(v,mkset nth(u,m))=1|)
    (label mm2)

3.  (assume |i<length v|)
    (label mm3)

4.  (trw |nth(v,i) ∈ mklset v| (open epsilon mklset)
         (use * nthmember mode: exact) )
    ;NTH(V,I)∈MKLSET(V)

5.  (rw * (use mm1 mode: exact direction: reverse))
    ;NTH(V,I)∈MKLSET(U)

6.  (rw * (use mklset,fact mode: exact) (open epsilon)
    ;∃K.K<LENGTH U∧NTH(U,K)=NTH(V,I)

7.  (define mv |mv<length u ∧nth(u,mv)=nth(v,i)| * )
    (label mm4)
    ;MV is unknown.
    ;the symbol MV is given the same declaration as M
    ;deps: (MM1 MM3)

a.  (ue (m mv) mm2 (use * mode: always))
    ;MULT(V,MKSET(NTH(V,I)))=1
    ;deps: (MM1 MM2 MM3)

9.  (ci mm3)
    ;I<LENGTH V⊃MULT(V,MKSET(NTH(V,I)))=1
    ;deps: (MM1 MM2)

10. (ci (mm1 mm2))
    ;MKLSET(U)=MKLSET(V)∧(∀M.M<LENGTH U⊃MULT(V,MKSET(NTH(U,M)))=1)⊃
    ;(I<LENGTH V⊃MULT(V,MKSET(NTH(V,I)))=1)
    (label mult_mult) ■
```

4.3.5. The Main Result for Association Lists: Every Permutation is an Injection.

The derivation of main result for alists follows.

Theorem (*Permutp Injectp*)

```
∀ALIST.PERMUTP(ALIST)⊃INJECTP(ALIST)
```

Proof.

```
    (proof permutp_injectp)

1.  (assume lpermutp alist l>
    (label permutp_injectp1)

2.  (rw * (open permutp))
    ;FUNCTP(ALIST)∧MKLSET(DOM(ALIST))=MKLSET(RANGE(ALIST))
    (label permutp_injectp2)

3.  (rw * (open functp))
    ;UNIQUENESS(DOM(ALIST))∧MKLSET(DOM(ALIST))=MKLSET(RANGE(ALIST))
    (label permutp_injectp3)
```

First step: disjointness of a suitable sequence of sets (Lemma *Inj Disj*)

```
    ;labels:UNIQUENESS,INJECTIVITY
    ;∀U.UNIQUENESS(U)≡INJ(U)

    ;labels: INJ,DISJ
    ;∀U.INJ(U)⊃DISJOINT(λM.MKSET(NTH(U,M)),LENGTH U)

4.  (derive |inj(dom(alist))| (* uniqueness_injectivity))
    ;deps: (PERMUTP_INJECTP1)

5.  (derive |DISJOINT(λM.MKSET(NTH(DOM(ALIST),M)),LENGTH (DOM(ALIST)))|
            (* inj,disj))
    (label permutp_injectp4)
```

Second step: multiplicity of the sets in the sequence is positive (*Permutp Injectp Lemma*)

```
    ;labels:PERMUTP,INJECTP-LEMMA
    ;∀U V.MKLSET(U)=MKLSET(V)⊃(∀M.M<LENGTH U⊃1≤MULT(V,MKSET(NTH(U,M))))

6.  (ue ((u.|dom alist|)(v.|range alist|)) permutp,injectp,lemma
        (permutp_injectp3 permutp_injectp4))
    ;∀M.M<LENGTH (DOM(ALIST))⊃1≤MULT(RANGE(ALIST),MKSET(NTH(DOM(ALIST),M)))
    (label permutp_injectp5)
```

Now we apply the Pigeon Hole principle:

```
;labels:PIGEONLIST
;∀SETSEQ U.DISJOINT(SETSEQ,LENGTH U)∧
;(∀K.K<LENGTH U⊃1≤MULT(U,SETSEQ(K)))⊃
;                (∀K.K<LENGTH U⊃1=MULT(U,SETSEQ(K))) I)

 ;need also
;labels:DOMRANGELENGTH
;∀ALIST.LENGTH (DOM(ALIST))=LENGTH (RANGE(ALIST))
```

```
7.  (ue ((setseq.|λm.mkset nth(dom alist,m)|)(u.|range alistl)) pigeonlist
      (use domrangelength mode: exact direction: reverse)
      permutp,inj ectp4 permutp,inj ectp5)
    ;∀K.K<LENGTH (DOM(ALIST))⊃
    ;l=MULT(RANGE(ALIST),MKSET(NTH(DOM(ALIST),K)))
```

Third step: injectivity (using lemmata *Mult Mult* and *Mult Inj*)

```
;labels:MULT,MULT
;∀U V.MKLSET(U)=MKLSET(V)∧
;     (∀K.K<LENGTH U⊃MULT(V,MKSET(NTH(U,K)))=1)⊃
;     (∀I.I<LENGTH V⊃MULT(V,MKSET(NTH(V,I)))=1)
```

```
8.  (ue ((u.|dom(alist)|)(v.|range(alist)|)) mult_mult
     ` permutp_injectp3 * )
    ;∀I.I<LENGTH (RANGE(ALIST))⊃
    ;MULT(RANGE(ALIST>,MKSET(NTH(RANGE(ALIST),I)))=l
    ;deps: (PERMUTP_INJECTP1)

    ;labels:MULT,INJ
    ;∀V.(∀K.K<LENGTH V⊃MULT(V,MKSET(NTH(V,K)))=1)⊃INJ(V)

9.  (ue (v lrange alist|) mult_inj * )
    ;INJ(RANGE(ALIST))
    ;deps: (PERMUTP_INJECTP1)

10. (derive |uniqueness(range alist)| (* uniqueness,injectivity))
    ;deps: (PERMUTP_INJECTP1)

11. (derive |injectp alistl (permutp_injectp2 *)(open injectp))
    ;deps: (PERMUTP_INJECTP1)

12. (ci (permutp,injectpl))
    ;PERMUTP(ALIST)⊃INJECTP(ALIST)
    (label permutp_injectp) ■
```

4.4. Application of the Pigeon Hole Principle to Lists of Numbers.

In the second application we give the proof of the theorem:

Any map of a finite set onto itself is 1-1,

by representing functions as lists of numbers

Here have a list u of numbers less than length(u) (*intoness*) and we know that every n less than length(u) occurs in u *(ontoness)*. This simplifies our problem. First we can consider for each m < length(u) the set $\{x : x = m\}$. The proof that the sequence **λm.mkset(m)** is disjoint is fairly straightforward. The second step-the proof that for m < length(u) the multiplicity of the set $\{x : x = m\}$ is positive-is immediate; only the third step -to prove $inj(u)$ assuming that the multiplicity of every such set in u is exactly 1-requires some work.

4.4.1. Step 1: Disjointness.

Lemma 4.4. *(Disjoint Number)* VN.DISJOINT(XXV.MKSET(XV),N)

Proof. First a useful fact: if $m \in \bigcup_{i<n}\{i\}$, then $m < n$.

```
    (proof disjoint-number)
1.  (ue (a |λn.∀m.(un((λxv.mkset(xv)),n))(m)⊃m<n|)
        proof-by-induction
        (part 1 (open mkset un emptyset union))
        (use normal mode: always)
        (use successor1 transitivity-of-order))
    ;∀N M.(UN(λXV.MKSET(XV),N))(M)⊃M<N
    (label dnl)
```

Therefore

$$\bigcup_{i<n}\{i\} \cap \{n\} = 0$$

and so. by induction on $n, \bigcup_{i<n}\{i\}$ is disjoint.

```
2.  (ue ((n.n)(m.n)) dnl irreflexivity,of-order)
    ;¬(UN(λXV.MKSET(XV),N))(N)

3.  (trw |(un(λyv.mkset(yv),n))(xv)∧(mkset(n))(xv)| *
         (part 2 (open mkset)))
    ;¬((UN(λYV.MKSET(YV),N))(XV)∧(MKSET(N))(XV))

4.  (ue (a |λn.disjoint(λxv.mkset(xv),n)|) proof-by-induction
        (open disjoint disj,pair emptyp intersection)
        (use * mode: exact))
    ;∀N.DISJOINT(λXV.MKSET(XV),N)
    (label disjoint-number) ■
```

This completes step 1.

4.4.3. Step 2: Ontoness Implies Multiplicity.

Lemma 4.5. (Onto *Mult*)

```
∀U.ONTO(U)⊃
   (∀N.N<LENGTH(U)⊃1≤MULT(U,MKSET(N)))
(label onto,mult) ■
```

This is immediate from the definition of onto and the lemma. *Member Mult*.

4.4.3. Step 3: Intoness Implies Multiplicity.

Using the lemma *Pigeonlist*. steps 1 and 2 we will conclude that

```
PERM(U)⊃(∀K.K<LENGTH U⊃1=MULT(U,MKSET(K)))
```

Let's look at the last step.

Lemma 4.6. (Into *Mult*)

```
∀U.INTO(U)∧(∀K.K<LENGTH U⊃1=MULT(U,MKSET(K)))⊃
   (K<LENGTH U⊃1=MULT(U,MKSET(NTH(U,K))))
```

Proof. Assume into(u) and that for all k < length(u) the multiplicity of the set $\{x : x = k\}$ is exactly 1.

```
    (proof into,mult)

1.  (assume |into(u)|)
    (label im1)

2.  (assume |∀k.k<length u⊃1=mult(u,mkset k)|)
    (label im2)

3.  (assume |k<length u|)
    (label im3)

4.  (rw iml (open into))
    ;∀N.N<LENGTH U⊃NATNUM(NTH(U,N))∧NTH(U,N)<LENGTH U
    ;deps: (IM1)
```

By *intoness*. nth(u,k) is a number less than length(u). The result is immediate from line 2.

```
5.  (ue (k |nth(u,k)|) im2 (use im3 * mode: exact))
    ;1=MULT(U,MKSET(NTH(U,K)))
    ;deps: (IM1 IM2 IM3)

6.  (ci im3)
    ;K<LENGTH U⊃1=MULT(U,MKSET(NTH(U,K)))
    ;deps: (IM1 IM2)

7.  (ci (iml im2))
    ;INTO(U)∧(∀K.K<LENGTH U⊃1=MULT(U,MKSET(K)))⊃
    ;(K<LENGTH U⊃1=MULT(U,MKSET(NTH(U,K))))
    (label into_mult) ■
```

4.4.4. The Main Result for Lists: Every Permutation is an Injection.

We now give the main result for functions represented as lists of numbers.

Theorem (*Perm Injectivity*) ∀U.PERM(U)⊃INJ(U)

Proof.

```
     (proof perm,inj)

 1.  (assume |perm u|)(label perm,injl)

 2.  (rw * (open perm onto))
     ;INTO(U)∧(∀N.N<LENGTH U⊃MEMBER(N,U))
     (label perm_inj2)
```

Second step: multiplicity is positive

```
     ;labels:MEMBER,MULT
     ;∀U Y A.MEMBER(Y,U)∧A(Y)⊃1≤MULT(U,A)

 3.  (ue ((u.u)(y.n)(a.|mkset n|)) member-mult
         (part 1 (open mkset)))
     ;MEMBER(N,U)⊃1≤MULT(U,MKSET(N))

 4.  (derive |∀n.n<length u⊃1≤mult(u,mkset n)| (perm_inj2 *))
     (label perm_inj3)
     ;deps: (PERM_INJ1)
```

Third step: the application of the pigeon hole

```
 5.  (ue ((setseq.|λxv.mkset(xv)|)(u.u))
         pigeonlist disjoint-number perm_inj3)
     ;∀K.K<LENGTH U⊃1=MULT(U,MKSET(K))
     (label perm_inj4)
     ;deps: (PERM_INJ1)

 6.  (ci perm_inj1)
     ;PERM(U)⊃(∀K.K<LENGTH U⊃1=MULT(U,MKSET(K)))
```

Fourth step: injectivity (using Lemmata INTO-MULT and MULT-INJ)

```
     ;labels:INTO,MULT
     ;∀U.INTO(U)∧(∀K.K<LENGTH U⊃1=MULT(U,MKSET(K)))⊃
     ;            (∀I.I<LENGTH U⊃1=MULT(U,MKSET(NTH(U,I))))

 7.  (derive |∀i.i<length u⊃1=mult(u,mkset(nth(u,i)))|
             (into-mult perm_inj2 *))
     ;deps: (PERM_INJ1)

     ;labels: MULT-INJ
     ;∀V.(∀K.K<LENGTH V⊃MULT(V,MKSET(NTH(V,K)))=1)⊃INJ(V)

 8.  ( u e  (v u) mult_inj * )
     ;INJ(U)
```

```
;deps: (PERM_INJ1)

(derive |inj(u)| (perm_injl perm_inj4 perm,inj,1emma))
;deps: (PERM_INJ1)
```

9. (ci perm,injl)
```
;PERM(U)⊃INJ(U)
```
. (label perm,injectivity) ■

5. Representation using Association Lists.

In this section we prove that permutations (over a finite domain) form a. group. using the representation of functions by association lists.

Remark. In this representation we do not need to restrict ourselves to functions from numbers to numbers: we may consider permutations of any finite set. However it is customary to view association lists as maps from atoms to S-espressions. We keep this convention.

To define our functions as maps from atoms to atoms would sligthly simplify some proofs below: notice that we need the assumption that all the members in the range are atoms in order to prove the lemmata *Invalistsort*, *Dom Invalist*, *Range Invalist* as well as Theorem 3 (ii) and (iii). In the case of permutations this condition is a consequence of the definition of permutation (as a map of a domain of atoms onto itself).

5.1. Definitions of Corn posit ion, Inverse and Identity.

The following functions and predicates on alists represent composition of functions, the identity function and the inverse of a function. Since the domain of our functions is not, fixed in advance. we must use a predicate rather than a function for identity.

```
     ;functions as association lists
     (proof assoc)

     ;composition of functions

1.   (decl (compalist) (infixname: |ꝏ|) (type: |ground⊗ground→ground|)
           (syntype: constant) (bindingpower: 930))

2.   (def ax compalist
        |∀alist1 alist2 xa y.nil ꝏ alist2=nil∧
                              ((xa.y) .alist1) ꝏ alist2=
                               (xa.appalist(y,alist2)) .(alist1 ꝏ alist2)|)
     (label compalistdef )

     ;the inverse function

3.   (decl invalist (type: GROUND→GROUND))

4.   (def ax invalist
        |∀alist  xa  y.invalist nil=nil∧
                      invalist((xa.y).alist)=(y.xa).invalist alist|)
     (label invalistdef )

     ;the identity function

5.   (decl idalistp (type: GROUND→TRUTHVAL))

6.   (defax idalistp
        |∀alist xa y.idalistp(nil)∧
                     (idalistp((xa.y).alist)≡xa=y∧idalistp alist) |)
```

```
(label idalistpdef)
```

Remark. In the present, section the reader should keep in mind that

$$\texttt{appalist(x,alist}_f \text{ m } \texttt{alist}_g\texttt{)}$$

represents $(g \circ f)(x)$. It would be helpful to use ieft notation $x(f\ g)$ for functions in our comments. but we do not want to change our notation just for one section.

5.2. Almost All the Facts.

We collect. here some Lemmata of general use. Their proof are remarkably short applications of alistinduction. The first two lemmata are used to check sorts for invalist and compalist.

```
        (proof alistf acts)

    1.  (ue (chi |λalist.alistp(alist ∞ alist1)|)
            alist induct ion
            (part 1 (open compalist)(use appalistsort mode: exact)))
        ;VALIST.ALISTP ALIST ∞ ALIST1
        (label simpinfo) (label compalistsort)      I

    2.  (ue (chi |λalist.allp(λx.atom x,range alist)⊃alistp invalist(alist) I)
            alistinduction
            (open range member invalist)
            (use allpf act
             ue: ((phi.|λx.atom x|)(x.y)(u.|range alist|)) mode: always) )
        ;∀ALIST.ALLP(λX.ATOM X,RANGE(ALIST))⊃ALISTP INVALIST(ALIST)
        (label simpinfo) (label invalistsort)      ■
```

We must consider with special care the behavior of the LISP function

$$\lambda\text{alist } \texttt{x.appalist(x,alist)}.$$

It is defined as `λalist x.cdr(assoc(x,alist))`, so it associates with x *the first y* such that `(x.y)` belongs to `alist` and has default value NIL, if there is no such y.

In Lemma 5.1, by assuming: that x belongs to `dom(alist)`, we ignore the default case: in Lemma 5.2, by taking into account the default case we prove equality instead of inclusion of domains.

Next, Lemma 5.3 proves that if x belongs to `dom(alist)`, then the value of `appalist(x,alist)` is not the default value NIL, but an element belonging to `range(alist)`: Lemma 5.4 says that if z belongs to `range(alist)`, then there is an x in `dom(alist)` such that appalist(x,alist) = z. Observe that this need not be true, unless `alist` represents a. function. i.e. unless `dom(alist)` has the *uniqueness* property. Indeed, if some (x .`z1`) occurs in `alist` before (x .`z`), with $z1 \neq z$, then appalist (x,alist) will give z1 as value.

Lemma 5.1 (*App Compalist*) $(g \circ f)(x) = g(f(x))$:

```
VALIST ALIST1 X.MEMBER(X,DOM(ALIST))⊃
   APPALIST(X,ALIST ∞ ALIST1)=APPALIST(APPALIST(X,ALIST),ALIST1)
```

Proof.

```
3.  (ue (chi |λalist.member(x,dom(alist))⊃
                    appalist(x,alist ∞ alistl)=
                    appalist(appalist(x,alist),alist1)|)
        alistinduction
        (part 1 (use appalistdef mode: always)
                (open dom member compalist assoc))
                (use normal mode: always))
    ;∀ALIST.MEMBER(X,DOM(ALIST))⊃
    ;      APPALIST(X,ALIST ∞ ALIST1)=APPALIST(APPALIST(X,ALIST),ALIST1)
    (label app,compalist) (label alist_lemma1) ■
```

The following Lemma says that the domain of $g \circ f$ is a subset of the domain of f.

Lemma *5.2* (*Dom Compalist*)

```
VALIST ALISTl.DOM(ALIST m ALIST1)=DOM(ALIST)
```

Proof.

```
4.  (ue (chi |λalist.dom(alist ∞ alist1)=dom(alist)|)
        alistinduction
        (open compalist dom))
    ;∀ALIST.DOM(ALIST ∞ ALISTl)=DOM(ALIST)
    (label dom_compalist)(label alist_lemma2) ■
```

The nest two Lemmata will be used in the proof of Theorem l(i).

Lemma 5.3 says that if x belongs to the domain of f then there is a $y = f(x)$ that belongs to the range of f.

Lemma 5.3 (*Nonempty Range*)

```
VALIST X.MEMBER(X,DOM ALIST)⊃
        (∃Y.MEMBER(Y,RANGE ALIST)∧APPALIST(X,ALIST)=Y)
```

The argument is by induction on `alists`.

```
(proof alist_lemma3)

1.  (ue (chi |λalist.member(x,dom alist)⊃
                    somep(λy.appalist(x,alist)=y,range alist)|)
        alistinduction
        (part 1 (open dom somep range member appalist assoc))
        (use normal mode: always))
    ;∀ALIST.MEMBER(X,DOM(ALIST))⊃SOMEP(λY.APPALIST(X,ALIST)=Y,RANGE(ALIST))

2.  (rw * (use somepfact mode: exact))
    ;∀ALIST.MEMBER(X,DOM(ALIST))⊃
    ;      (∃X1.MEMBER(X1,RANGE(ALIST))∧APPALIST(X,ALIST)=X1)
    (label nonempty,range) (label alist_lemma3) ■
```

Remark. Example 7. We prove first the formula, in line 1 (containing the recursively defined predicate samep instead of the existential quantifier as in line 2). In this way we considerably shorten the proof. Let's analyze the proof and see how the rewriter of EKL simulates it.

The induction step of line 1 is

```
;(∀XA Y ALIST.(MEMBER(X,DOM(ALIST))⊃
;               SOMEP(λY2.APPALIST(X,ALIST)=Y2,RANGE(ALIST)))⊃
;              (MEMBER(X,DOM((XA.Y).ALIST))⊃
;               SOMEP(λY1.APPALIST(X,(XA.Y).ALIST)=Y1,RANGE((XA.Y).ALIST))))
```

By expanding

`member(x,dom((xa.y).alist))`

we obtain two cases:

(i) `x=xa`, in which case `cdr(assoc(x,(xa,y).alist))` is y, and y is clearly a member of `range((xa.y).alist)`;

(ii) member (x ,dom `alist`). In this case the induction hypothesis yields `somep(λy2.appalist(x,alist)=y2,range(alist))`.

(The two cases are dealt with separately by using as a. rewriter the formula

`∀p q r. (p∨q⊃r)≡(p⊃r)∧(q⊃r)`

labeled NORMAL, as we saw in previous examples.)

Consider how the rewriting process accomplishes this inference. By expanding appalist and assoc in

(⋆) `SOMEP(λY1.APPALIST(X,(XA.Y).ALIST)=Y1,RANGE((XA.Y).ALIST))`

we have:

```
;the term APPALIST(X,(XA.Y).ALIST) is replaced by:
CDR ASSOC(X,(XA.Y).ALIST>
;the term ASSOC(X,(XA.Y).ALIST) is replaced by:
IF X=XA THEN XA.Y ELSE ASSOC(X,ALIST)
```

Now the conditional term is 'pushed outside' the function cdr:

```
;the term CDR (IF X=XA THEN XA.Y ELSE ASSOC(X,ALIST)) is replaced by:
IF X=XA THEN CDR (XA.Y) ELSE CDR ASSOC(X,ALIST)
;the term CDR (XA.Y) is replaced by:
Y
;the term (IF X=XA THEN Y ELSE CDR ASSOC(X,ALIST))=Y1 is replaced by:
IF X=XA THEN Y=Y1 ELSE CDR ASSOC(X,ALIST)=Y1
```

Nest by expanding range we obtain:

```
;the term RANGE((XA.Y).ALIST) is replaced by:
Y.RANGE(ALIST)
```

Now somep is expanded:

```
;the term
SOMEP(λY1.(IF X=XA THEN Y=Y1 ELSE CDR ASSOC(X,ALIST)=Y1),Y.RANGE(ALIST))
is replaced by:
IF (IF X=XA
       THEN Y=Y
       ELSE CDR ASSOC(X,ALIST)=Y)
   THEN TRUE
   ELSE SOMEP(λY1.(IF X=XA
                      THEN Y=Y1
                      ELSE CDR ASSOC(X,ALIST)=Y1),RANGE(ALIST))
```

In the innermost conditional

```
;the term Y=Y is replaced by:
TRUE
```

so that the 'if' case of the outer conditional becomes

```
;the term IF X=XA THEN TRUE ELSE CDR ASSOC(X,ALIST)=Y is replaced by:
X=XA∨CDR ASSOC(X,ALIST)=Y
```

On the other hand in the 'else' case of outer conditiona.

```
;the term X=XA is replaced by:
FALSE
;the term IF FALSE THEN Y=Y1 ELSE CDR ASSOC(X,ALIST)=Y1 is replaced by:
CDR ASSOC(X,ALIST)=Y1
```

In conclusion. the term (⋆) becomes

```
X=XA∨CDR ASSOC(X,ALIST)=Y∨SOMEP(λY1.CDR ASSOC(X,ALIST)=Y1,RANGE(ALIST))
```

and it is this formula that rewrites to true in both cases (i) and (ii). □

Similarly, Lemma 5.4 says that if z belongs to the range of f then there is an x in the domain of f such that $f(x)$ = 3. The proof is left to the Appendis.

Lemma *5.4 (Nonempty Domain)*

```
VALIST Z.UNIQUENESS DOM(ALIST)∧MEMBER(Z,RANGE ALIST)⊃
        (∃X.MEMBER(X,DOM ALIST)∧APPALIST(X,ALIST)=Z)
(label alist_lemma4)
```

The following Lemma, (describing the behavior of compalist with respect to the second alist) is used in the induction step of the proofs of theorems 3(ii) and 3(iii).

```
; compalist lemma

5. (ue (chi |λalist.¬member(za,range alist)⊃
                alist ∞ ((za.z).alist1)=alist ∞ alist1|)
     alistinduction
     (open member range compalist assoc) (use demorgan mode: always))
   ;∀ALIST.¬MEMBER(ZA,RANGE(ALIST))⊃ALIST ∞ ((ZA.Z).ALIST1)=ALIST ∞ ALIST1
   (label compalist-lemma) ■
```

We easily check that samemap guarantees identity of composition on the right: (*Samemap Right*)

```
6.  (ue (chi |λalist.samemap(alist1,alist2)⊃alist ∞ alist1=alist ∞ alist2|)
        alistinduction
        (part 1 (open compalist samemap)))
    ;∀ALIST.SAMEMAP(ALIST1,ALIST2)⊃ALIST ∞ ALIST1=ALIST ∞ ALIST2
    (label simpinfo) (label samemap_right)    ■
```

When composing on the left. the best possible analogue is the following: (*Samemap Left*)

```
    ∀ALIST ALIST1 ALIST2.SAMEMAP(ALIST1,ALIST2)⊃
                              SAMEMAP(ALIST1 ∞ ALIST,ALIST2 ∞ ALIST)
```

The proof uses Lemmata 5.1 and 5.2 and is left to the Appendix.

The main property of the identity alist is given by the following:

Lemma 5.5 (*Main Idalistp*)

```
    ∀ALIST Y.IDALISTP(ALIST)∧MEMBER(Y,DOM(ALIST))⊃CDR ASSOC(Y,ALIST)=Y
```

Proof.

```
7.  (ue (chi |λalist.idalistp(alist)∧member(y,dom alist)⊃
        appalist(y,alist)=y|)
        alistinduction
        (open idalistp appalist assoc member dom) (use normal mode: always))
    ;∀ALIST.IDALISTP(ALIST)∧MEMBER(Y,DOM(ALIST))⊃CDR ASSOC(Y,ALIST)=Y
    (label idalistp,main) ■
```

Finally, we prove two lemmata essential for the proof of Theorem 3.
We show that dom(invalist) is the same as range and that range(invalist) is dom.

```
    ;dom invalist

8.  (ue (chi |λalist.allp(λx.atom x,range alist)⊃
                  dom invalist(alist)=range alist|
        alistinduction
        (open dom range invalist) (use invalistsort)
        (use allpfact
         ue: ((phi.|λx.atom x|)(x.y)(u.|range alist|)) mode: always) )
    ;∀ALIST.ALLP(λX.ATOM X,RANGE(ALIST))⊃
    ;       DOM(INVALIST(ALIST))=RANGE(ALIST)
    (label dom,invalist) ■

    ;range invalist

9.  (ue (chi |λalist.allp(λx.atom x,range alist)⊃
                  range invalist(alist)=dom alist|)
        alistinduction
        (open dom range invalist) (use invalistsort)
        (use allpfact
         ue: ((phi.|λx.atom x|)(x.y)(u.|range alistl)) mode: always) )
    ;∀ALIST.ALLP(λX.ATOM X,RANGE(ALIST))⊃
    ;       RANGE(INVALIST(ALIST))=DOM(ALIST)
    (label range-invalist) ■
```

5.3. The Composition of Permutations is a Permutation.

We want to prove that if two alists, `alist` and `alist1` are permutp, then also their composition `alist` ∘ `alist1` is a permutp: i.e. (by the definition of permutp) we know that the domains of `alist` and `alist1` have the *uniqueness* property and their 'domains and 'ranges' are the same set and we want to show that

(i) *uniqueness* holds of `dom(alist ∘ alist1)`;

(ii) the dom and the range of `(alist ∘ alist1)` are the same set.

To prove (i) it is enough to show that the `dom(alist ∘ alist1)` is the same as `dom(alist)`. To prove (ii) we prove inclusion in both directions.

The proof of (ii) is the longest in this section. The reason is that we cannot use induction on alists in proving facts about the range of `dom(alist ∘ alist1)` *as a* set.

Theorem 1 (i) (*Permutp Compalist*)

```
∀ALIST ALIST1.PERMUTP(ALIST)∧PERMUTP(ALIST1)∧
              MKLSET(DOM(ALIST))=MKLSET(DOM(ALIST1))⊃
              PERMUTP(ALIST ∘ ALIST1)
```

This is proved through a main Lemma:

Lemma *Range Compose*, part 1:

```
∀ALIST ALIST1.PERMUTP(ALIST)∧
              MKLSET(DOM(ALIST))=MKLSET(DOM(ALIST1))⊃
              MKLSET(RANGE(ALIST∘ALIST1))⊂MKLSET(RANGE(ALIST1))
```

Lemma *Range Compose*, part 2:

```
∀ALIST ALIST1.PERMUTP(ALIST)∧PERMUTP(ALIST1)∧
              MKLSET(DOM(ALIST))=MKLSET(DOM(ALIST1))⊃
              MKLSET(RANGE(ALIST1))⊂MKLSET(RANGE(ALIST ∘ ALIST1))
```

5.3.1. Proof Range Compose, First Part.

In Part 1 we show that if `permutp(alist)` and mklset `dom(alist)=mklset dom(alist1)`, then `range(alist ∘ alist1)` is a subset of `range(alist1)`.

Let f and g be the functions represented by `alist` and `alist1`, respectively. The argument can be summarized as follows: given z in the range of $g \circ f$, choose an element x_z in the inverse image of z by $g \circ f$. Such element is in the domain of f. By definition of composition, if $z = (g \circ f)(x_z)$, then $z = g(f(x_z))$, so z belongs to the range of g.

```
(proof range-compose)

1.  (assume |permutp(alist)|)
    (label rcl)

2.  (rw * (open permutp functp))
    ;UNIQUENESS(DOM(ALIST))∧MKLSET(DOM(ALIST))=MKLSET(RANGE(ALIST))
    (label rc2)
    ;deps: (RC1)
```

The nest line says that the functions f and g represented by **alist** and **alist** 1 have the same domain.

```
3.  (assume |mklset dom(alist)=mklset dom(alist1)|)
    (label rc3)

4.  (assume |member(z,range(alist m alist1))|)
    (label rc4)
```

By applying Lemma 4 we associate to z an element x_z in dom(alist m alistl). By Lemma. 2 x_z lies in dom(alist) (line 6).

```
5.  (ue ((alist.|alist ∞ alist1|)(z.z)) nonempty,domain
     · (use dom,compalist rc2 rc4 mode: exact) )
    ;∃X.MEMBER(X,DOM(ALIST))∧APPALIST(X,ALIST ∞ ALIST1)=Z
    ;deps: (RC1 RC4)

6.  (define xxvv
            |member(xxvv,dom alist)∧appalist(xxvv,alist ∞ alist1)=z| * )
    (label rc5)
    ;deps: (RC1 RC4)
```

Apply Lemma. 1:

```
7.  (rw * (use app,compalist mode: always))
    ;MEMBER(XXVV,DOM(ALIST))∧APPALIST(APPALIST(XXVV,ALIST),ALIST1)=Z
    (label rc6)
    ; deps: (RC1 RC4)
```

This represents the fact that if $z = (g \circ f)(x_z)$, then $z = g(f(x_z))$. The proof is not finished. however. We have to check that the two applications of appalist do not give default value.

By applying lemma, 3, we associate to x_z its image y_z in the range(alist).

```
8.  (define yyvv |member(yyvv,range alist)∧appalist(xxvv,alist)=yyvv|
            (nonempty,range rc6))
    (label rc7)
    ;deps: (RC1 RC4)

9.  (trw |yyvv ∈ mklset range(alist)| (open mklset epsilon) rc7)
    ;YYVV∈MKLSET(RANGE(ALIST))
    ;deps: (RC1 RC4)
```

By the assumption of line **3** we know that y_z belongs to dom(alist1).

```
10. (rw * (use rc2 mode: exact direction: reverse)
          (use rc3 mode: exact))
    ;YYVV∈MKLSET(DOM(ALIST1))
    ;deps: (RC1 RC3 RC4)

11. (rw * (open epsilon mklset))
    ;MEMBER(YYVV,DOM(ALISTl))
    ;deps: (RC1 RC3 RC4)
```

We apply again lemma **3** to pick the image of y_z in range(alist1).

```
12. (define zzvv |member(zzvv,range alist1)∧appalist(yyvv,alist1)=zzvv|
            (nonempty-range * ))
    (label rc8)
    ;deps: (RC1 RC3 RC4)
```

By lines 7 and 8 such an image is z.

```
13. (rw rc6 rc7)
    ;MEMBER(XXVV,DOM(ALIST))∧APPALIST(YYVV,ALIST1)=Z
    ;deps: (RC1 RC4)

14. (trw |zzvv=z| * (use rc8 mode: always direction: reverse))
    ; zzvv=z
    ;deps: (RC1 RC3 RC4)
```

Hence. z is in the range(alist1).

```
15. (trw |member(z,range alist1)| rc8
         (use * mode: exact direction: reverse))
    ;MEMBER(Z,RANGE(ALIST1))
    ;deps: (RC1 RC3 RC4)

16. (ci rc4)
    ;MEMBER(Z,RANGE(ALIST ∘ ALIST1))⊃MEMBER(Z,RANGE(ALIST1))
    ;deps: (RC1 RC3)

17. (trw |mklset range(alist ∘ alist1)⊂mklset range(alist1) | *
         (open mklset inclusion))
    ;MKLSET(RANGE(ALIST ∘ ALIST1))⊂MKLSET(RANGE(ALIST1))
    ;deps: (RC1 RC3)

18. (ci (rcl rc3))
    ;PERMUTP(ALIST)∧MKLSET(DOM(ALIST))=MKLSET(DOM(ALIST1))⊃
    ;MKLSET(RANGE(ALIST ∘ ALIST1))⊂MKLSET(RANGE(ALIST1))
    (label range-compose)
```

5.3.2. Proof of Range Compose, Second Part.

In Part 2, again under the assumption that

permutp(alist) ∧ mklset dom(alist)=mklset dom(alist1),

we prove that

range(alist) ⊂ range(alist ∘ alist1)

The derivation represents the following argument: suppose that f and g are maps of the same finite set onto itself and z belongs to the range of g. If y_z is in the inverse image of z by g, then y_z is in the range of f. Moreover, if x_z is in the inverse image of y_z by f, then $z = g(f(x_z))$, i.e. $z = (g \circ f)(x_z)$. Therefore z is in the range of $g \circ f$.

```
    (proof range-compose2)

1   (assume |permutp(alist)|)
    (label rc21)

2   (rw * (open permutp functp))
    ;UNIQUENESS(DOM(ALIST))∧MKLSET(DOM(ALIST))=MKLSET(RANGE(ALIST))
    (label rc22)

3   (assume |permutp(alist1)|)
    (label rc23)

4   (rw * (open permutp functp))
    ;UNIQUENESS(DOM(ALIST1))∧MKLSET(DOM(ALIST1))=MKLSET(RANGE(ALIST1))
    (label rc24)

5   (assume |mklset dom(alist)=mklset dom(alist1)|)
    (label rc25)

6.  (assume |member(z,range alist1)|)
    (label rc26)
```

Given z in $range(alist1)$, using lemma -4 we pick a y_z in $dom(alist1)$ such that

appalist(y_z,alist1) = z.

```
7.  (define yv1 |member(yv1,dom alist1)∧appalist(yv1,alist1)=z|
            (nonempty,domain rc24 rc26))
    (label rc27)

8.  (trw |yv1 ∈ mklset dom(alist1)| * (open epsilon mklset))
    ;YV1∈MKLSET(DOM(ALIST1))
    ;deps: (RC23 RC26)
```

By our assumptions y_z is in range(alist).

```
9.  (rw * (use rc25 mode: exact direction: reverse)
          (use rc22 mode: exact))
    ;YV1∈MKLSET(RANGE(ALIST))
    ;deps: (RC21 RC23 RC25 RC26)

10. (rw * (open epsilon mklset))
    ;MEMBER(YV1,RANGE(ALIST))
    (label rc28)
    ;deps: (RC21 RC23 RC25 RC26)
```

By applying again lemma 4 we can pick x_z in dom(alist) such that

$$\texttt{appalist}(x_z,\texttt{alist1}) = y_z.$$

```
11. (define xv1 |member(xv1,dom alist)∧appalist(xv1,alist)=yv1|
            (nonempty-domain rc22 rc28))
    (label rc29)
    ;deps: (RC21 RC23 RC25 RC26)
```

Apply lemma 2:

```
12. (trw |member(xv1,dom(alist ∞ alist1))| * (use dom_compalist))
    ;MEMBER(XV1,DOM(ALIST ∞ ALIST1))
    (label rc30)
    ;deps: (RC21 RC23 RC25 RC26)
```

Now, by rewriting, we derive

$$z = \texttt{appalist}(\texttt{appalist}(x_z,\texttt{alist}),\texttt{alist1}) = \texttt{appalist}(x_z,\texttt{alist} \infty \texttt{alist1}).$$

```
13. (trw |appalist(xv1,alist ∞ alist1)| rc29 rc30
          (use app_compalist rc29 rc27 mode: always))
    ;APPALIST(XV1,ALIST ∞ ALIST1)=Z
    (label rc31)
    ;deps: (RC21 RC23 RC25 RC26)
```

We have to check that z is not the default value of appalist. We apply *Nonempty Range*:

```
14. (ue ((alist.|alist ∞ alist1|)(x.xv1)) nonempty,range
         (use dom_compalist rc22 rc30 mode: always))
    ;∃Y.MEMBER(Y,RANGE(ALIST ∞ ALIST1))∧APPALIST(XV1,ALIST m ALIST1)=Y
    ;deps: (RC21 RC23 RC25 RC26)

15. (define zvl |member(zv1,range(alist ∞ alist1))∧
                 appalist(xv1,alist ∞ alist1)=zv1 | * )
    (label rc32)
    ;deps: (RC21 RC23 RC25 RC26)

16. (trw |zv1=z| rc31 (use * mode: always direction: reverse))
    ;ZV1=Z
    ;deps: (RC21 RC23 RC25 RC26)
```

So z belongs to the range(alist m alistl).

```
17. (trw |member(z,range(alist ∞ alist1))| rc32
        (use * mode: exact direction: reverse))
    ;MEMBER(Z,RANGE(ALIST m ALISTl))
    ;deps: (RC21 RC23 RC25 RC26)

18. (ci rc26)
    ;MEMBER(Z,RANGE(ALIST1))⊃MEMBER(Z,RANGE(ALIST m ALISTl))
    ;deps: (RC21 RC23 RC25)

19. (trw |mklset range(alist1)⊂mklset range(alist ∞ alist1)| *
        (open inclusion mklset) )
    ;MKLSET(RANGE(ALIST1))⊂MKLSET(RANGE(ALIST m ALISTl))
    ;deps: (RC21 RC23 RC25)

20. (ci (rc21 rc23 rc25))
    ;PERMUTP(ALIST)∧PERMUTP(ALIST1)∧MKLSET(DOM(ALIST))=MKLSET(DOM(ALIST1))⊃
    ;MKLSET(RANGE(ALIST1))⊂MKLSET(RANGE(ALIST ∞ ALISTl))
    (label range-compose)
```

5.3.3. Conclusion of the Proof of Permutp Compose.

Npw we conclude the theorem: by Lemma *Range Compose* and estensionality we show that

$$\text{mklset(range(alist m alistl))} = \text{mklset(range(alist1))}$$

(line 7). By the definition of permutp and the assumption that the alistl and alist2 are premutations of the same set (line 3), mklset range(alist1) is equal to mklset dom(alist). An application of Lemma 2 (line 10) is enough to reach the conclusion.

```
    (proof permutp,compalist)

1.  (assume |permutp(alist)|)
    (label permut,compl)

2.  (assume |permutp(alist1)|)
    (label permut_comp2)

3.  (assume |mklset(dom(alist))=mklset(dom(alist1))|)
    (label permut_comp3)

4.  (derive |mklset(range(alist m alist1))⊂mklset(range(alist1))∧
            mklset(range(alist1))⊂mklset(range(alist m alist1))|
             (permut,compl permut_comp2 permut_comp3 range-compose))
    ;deps: (PERMUT_COMP1 PERMUT_COMP2 PERMUT_COMP3)

5.  (rw * (open inclusion))
    ;(∀XV.(MKLSET(RANGE(ALIST ∞ ALIST1)))(XV)⊃(MKLSET(RANGE(ALIST1)))(XV))∧
    ;(∀XV.(MKLSET(RANGE(ALIST1)))(XV)⊃(MKLSET(RANGE(ALIST m ALIST1)))(XV))
    ;deps: (PERMUT_COMP1 PERMUT_COMP2 PERMUT_COMP3)
```

```
6.  (derive |∀xv.(mklset range(alist m alist1))(xv)≡
                 (mklset range(alist1))(xv)| * )
```

Remember *Set Extensionality:*

```
;labels:SET,EXTENSIONALITY
;(AXIOM |∀A B.(∀XV.XV∈A≡XV∈B)⊃A=B|)

7.  (ue ((a.|mklset range(alist m alistl) |)
         (b.|mklset range(alist1)|))
        set,extensionality
        * (open epsilon))
    ;MKLSET(RANGE(ALIST m ALIST1))=MKLSET(RANGE(ALIST1))
    ;deps: (PERMUT_COMP1 PERMUT_COMP2 PERMUT_COMP3)
    (label permut_comp4)

8.  (rw permut,compl (open permutp functp))
    ;UNIQUENESS(DOM(ALIST))∧MKLSET(DOM(ALIST))=MKLSET(RANGE(ALIST))
    (label permut_comp5)

9.  (rw permut_comp2 (open permutp))
    ;FUNCTP(ALIST1)∧MKLSET(DOM(ALIST1))=MKLSET(RANGE(ALIST1))

10. (trw |uniqueness(dom(alist m alist1))∧
          mklset dom(alist m alist1)=mklset range(alist m alist1)|
         (use dom,compalist permut_comp4 mode: exact) permut_comp5
         (use * permut_comp3 mode: always direction: reverse))
    ;UNIQUENESS(DOM(ALIST m ALIST1))∧
    ;MKLSET(DOM(ALIST m ALIST1))=MKLSET(RANGE(ALIST m ALISTl))
    ;deps: (PERMUT_COMP1 PERMUT_COMP2 PERMUT_COMP3)

11. (trw |permutp(alist m alist1)| * (open permutp functp))
    ;PERMUTP(ALIST m ALISTl)
    ;deps: (PERMUT_COMP1 PERMUT_COMP2 PERMUT_COMP3)

12. (ci (permut_comp1 permut_comp2 permut_comp3))
    ;PERMUTP(ALIST)∧PERMUTP(ALIST1)∧MKLSET(DOM(ALIST))=MKLSET(DOM(ALIST1))⊃
    ;PERMUTP(ALIST m ALISTl)
    (label permutp,compalist) ■
```

5.4. Associativity of Composition.

To show that composition is associative is very straightforward. Line 3 simply helps the rewriter in the inductive step to expand the antecedent of the induction formula (line 4).

Theorem 1 (ii) *(Compalist Associativity)*

```
∀ALIST ALISTl ALIST2.MKLSET(RANGE(ALIST))⊂MKLSET(DOM(ALIST1))⊃
                         ALIST m (ALISTl m ALIST2)=(ALIST m ALISTl) m ALIST2
```

Proof.

```
(proof compalist-associativity)

1. (trw |mklset(range((xa.y).alist))⊂mklset(dom alist1)⊃
        member(y,dom alist1)∧mklset range(alist)⊂mklset dom(alist1)|
       (open mklset inclusion range member)
       (use normal mode: always))
   ;MKLSET(RANGE((XA.Y).ALIST))⊂MKLSET(DOM(ALIST1))⊃
   ;MEMBER(Y,DOM(ALIST1))∧MKLSET(RANGE(ALIST))⊂MKLSET(DOM(ALIST1))

2. (trw |member(y,dom alist1)∧mklset range(alist)⊂mklset dom(alist1)⊃
        mklset(range((xa.y).alist))⊂mklset(dom alist1)| (der)
       (open mklset inclusion range member)
       (use normal mode: always))

3. (derive |mklset(range((xa.y).alist))⊂mklset(dom alist1)≡
            member(y,dom alist1)∧mklset range(alist)⊂mklset dom(alist1)|
          (* -2))
   (label helpinduction)

4. (ue (chi |λalist.mklset(range alist)⊂mklset(dom alist1)⊃
                 alist m (alistl m alist2)=(alist m alistl) m alist2|)
      alistinduction
      (part 1 (open compalist) (use app,compalist * mode: always)))
   ;∀ALIST.MKLSET(RANGE(ALIST))⊂MKLSET(DOM(ALIST1))⊃
   ,       ALIST m (ALIST1 m ALIST2)=(ALIST m ALIST1) ∞ ALIST2
   (label compalist-associativity) ■
```

5.5. The Identity Alist.

It is a simple matter to prove that an **alist** representing an identity function satisfies the property permutp.

Theorem 2 (i) (*Idalistp Permutp*)

∀ALIST.FUNCTP(ALIST)∧IDALISTP(ALIST)⊃PERMUTP(ALIST)

Proof.

```
1. (ue (chi |λalist.idalistp(alist)⊃dom alist=range alist|) alistinduction
      (open idalistp dom range)).
   ;∀ALIST.IDALISTP(ALIST)⊃DOM(ALIST)=RANGE(ALIST)

2. (trw |∀alist.functp(alist)∧idalistp(alist)⊃permutp(alist)|
       (open functp permutp)(use * mode: always))
   ;∀ALIST.FUNCTP(ALIST)∧IDALISTP(ALIST)⊃PERMUTP(ALIST)
   (label idalistp_permutp) ■
```

Using the same 'help for the rewriter that was used in the preceding section it is easy to prove the main theorem for right identity. We prove that if alistl represents the identity function i and alist represents a. function f (i and f being defined on the right domains), then alist ∞ alistl = alist, i.e. $i \circ f = f$.

Theorem 2 (ii) (*Idalistp Right)*

```
∀ALIST1.IDALISTP(ALIST1)⊃
(∀ALIST.MKLSET(RANGE(ALIST))⊂MKLSET(DOM(ALIST1))⊃ALIST ∞ ALISTl=ALIST)
```

Proof.

```
3.  (assume |idalistp(alist1)|)

4.  (ue (chi |λalist.mklset(range(alist))⊂mklset(dom(alist1))⊃
                      (alist ∞ alist1=alist)|)
        alistinduction
        (part 1 (open compalist))
        (use helpinduction idalistp,main * mode: always))
    ;∀ALIST.MKLSET(RANGE(ALIST))⊂MKLSET(DOM(ALIST1))⊃ALIST ∞ ALISTl=ALIST
    ;deps: (4)

5.  (ci -2)
    ;IDALISTP(ALIST1)⊃
    ;(∀ALIST.MKLSET(RANGE(ALIST))⊂MKLSET(DOM(ALIST1))⊃ALIST ∞ ALISTl=ALIST)
    (label idalistp-right) ■
```

Left identity presents a different kind of problem. Here we pay for our sins. namely for the fact that our representation is not unique. What we prove **is** that if alistid is idalistp then alistid ∞ alist is in the relation samemap with alist. The proof uses the main fact about identity alist (lemma. *Main Idalistp*).

Theorem 2 (iii) (*Left Idalistp*)

```
∀ALIST.IDALISTP(ALISTID)∧MKLSET(DOM(ALISTID))=MKLSET(DOM(ALIST))⊃
        SAMEMAP(ALISTID ∞ ALIST,ALIST)
```

Proof.

```
    (proof idalistp-left)

1.  (assume |idalistp alistid|)
    (label idal_l1)
    ;ALISTID is unknown.
    ;the symbol ALISTID is given the same declaration as ALIST

2.  (assume |mklset dom(alistid)=mklset dom(alist)|)
    (label idal_l2)

3.  (assume |y∈mklset(dom(alistid ∞ alist))|)
    (label idal_l3)

4.  (rw * (use dom_compalist mode: exact)(open epsilon mklset))
```

```
    (label idal_14)
    ;MEMBER(Y,DOM(ALISTID))
    ;deps: (IDAL_L3)

5.  (trw |appalist(y,alistid ∞ alist)| (use app,compalist * mode: exact))
    ;APPALIST(Y,ALISTID ∞ ALIST)=APPALIST(APPALIST(Y,ALISTID),ALIST)
    (label idal_15)

    ;labels: IDALISTP,MAIN
    ;∀ALIST Y.
    ; IDALISTP(ALIST)∧MEMBER(Y,DOM(ALIST))⊃APPALIST(Y,ALIST)=Y

6.  (derive |appalist(y,alistid)=y|
            (idalistp,main idal_11 idal_14))
    ;deps: (IDAL_L1 IDAL_L3)

7.  (rw idal_15 * )
    ;APPALIST(Y,ALISTID ∞ ALIST)=APPALIST(Y,ALIST)
    ;deps: (IDAL_L1 IDAL_L3)

8.  (ci idal_13)
    ;Y∈MKLSET(DOM(ALISTID ∞ ALIST))⊃
    ;APPALIST(Y,ALISTID ∞ ALIST)=APPALIST(Y,ALIST)
    (label idal_16)
    ;deps: (IDAL_L1)

-9. (trw |mklset(dom(alistid ∞ alist))=mklset dom(alist)|
         (use dom,compalist idal_12 mode: exact))
    ;MKLSET(DOM(ALISTID ∞ ALIST))=MKLSET(DOM(ALIST))
    ;deps: (IDAL_L2)

    ;labels:SAMEMAPDEF
    ;∀ALIST ALIST1.SAMEMAP(ALIST,ALIST1)≡
    ;             MKLSET(DOM(ALIST))=MKLSET(DOM(ALIST1))∧
    ;             (∀Y.Y∈MKLSET(DOM(ALIST))⊃
    ;                 APPALIST(Y,ALIST)=APPALIST(Y,ALIST1))

10. (trw |samemap(alistid ∞ alist,alist)| (open samemap) (idal_16 *))
    ;SAMEMAP(ALISTID ∞ ALIST,ALIST)
    ;deps: (IDAL_L1 IDAL_L2)

11. (ci (idal_11 idal_12))
    ;IDALISTP(ALISTID)∧MKLSET(DOM(ALISTID))=MKLSET(DOM(ALIST))⊃
    ;SAMEMAP(ALISTID ∞ ALIST,ALIST)
    (label idalistp,left) ■
```

5.6. Inverse of a Permutation is a Permutation.

We promised short proofs for the inverse operation using association lists. **Et** voila.!...

Theorem 3. (ii) *(Right Invalist)*

```
∀ALIST.ALLP(λX.ATOM X,RANGE(ALIST))∧INJECTP(ALIST)⊃
       IDALISTP(ALIST ∘ INVALIST(ALIST))
```

Proof.

```
    (proof invalist)

1.  (ue (chi |λalist.allp(λx.atom x,range(alist))∧injectp(alist)⊃
                 idalistp(alist ∘ invalist(alist))|)
        alistinduction
        (part 1
          (use allpfact
               ue: ((phi.|λx.atom x|)(x.y)(u.|range alistl)) )
          (open range injectp functp uniqueness
                invalist idalistp compalist appalist assoc)
          (use invalistsort dom,invalist compalist,lemma mode: exact)))
    ;∀ALIST.ALLP(λX.ATOM X,RANGE(ALIST))∧INJECTP(ALIST)⊃
    ,       IDALISTP(ALIST ∘ INVALIST(ALIST))
    (label invalist,right) ■
```

Theorem 3. (iii) (Left *Invalist*)

```
∀ALIST.ALLP(λX.ATOM X,RANGE(ALIST))∧INJECTP(ALIST)⊃
       IDALISTP(INVALIST(ALIST) ∘ ALIST)
```

Proof.

```
2.  (assume |allp(λx.atom x,range(alist))|)

3.  (ue ((alist.|invalist(alist)|)(alist1.|alist|)(za.xa)(z.y))
        compalist,lemma
        (use * invalistsort range,invalist mode: exact))
    ;¬MEMBER(XA,DOM(ALIST))⊃
    ;INVALIST(ALIST) ∘ ((XA.Y).ALIST)=INVALIST(ALIST) ∘ ALIST

4.  (ci -2)
    ;ALLP(XX.ATOM X,RANGE(ALIST))⊃
    ;(¬MEMBER(XA,DOM(ALIST))⊃
    ;    INVALIST(ALIST) ∘ ((XA.Y).ALIST)=INVALIST(ALIST) ∘ ALIST)

5.  (ue (chi |λalist.allp(λx.atom x,range(alist))∧injectp(alist)⊃
                 idalistp(invalist(alist) ∘ alist)|)
        alistinduction
        (part 1 (open allp range injectp functp uniqueness
                   invalist compalist appalist assoc idalistp)
        invalistsort (use range-invalist mode: exact) (use * mode: always)))
    ;∀ALIST.ALLP(λX.ATOM X,RANGE(ALIST))∧INJECTP(ALIST)⊃
    ;       IDALISTP(INVALIST(ALIST) ∘ ALIST)
    (label invalist,left) ■
```

Part (i) of Theorem 3 is also quite easy. We need first a Lemma, to make sure that dom(invalist(alist))

is made of atoms only. The proof of this lemma is a simple example of method of proof frequently used in this paper: first we prove a property of lists by a. convenient induction and then we derive a property of sets (i.e. we abstract from the order given by the list.).

Lemma (*Atomrange*)

```
∀ALIST.MKLSET(DOM(ALIST))=MKLSET(RANGE(ALIST))⊃
       ALLP(λX.ATOM X,RANGE(ALIST))
```

Proof.

```
    (proof atomrange)

1.  (assume |mklset(dom(alist))=mklset(range(alist))|)
    (label arl)

2.  (ue (chi |λalist.allp(λx.atom(x),dom alist)|)
             alistinduction
             (open allp dom))
    ;∀ALIST.ALLP(λX.ATOM X,DOM(ALIST))
    (label ar2)

3.  (ue ((phi1.|λx.atom(x)|)(x.x)(u.|dom alist|)) allp,elimination * )
    ;MEMBER(X,DOM(ALIST))⊃ATOM X

4.  (trw |mklset dom(alist)⊂(λx.atom x)| * (open inclusion mklset) )
    ;MKLSET(DOM(ALIST))⊂(λX.ATOM X)

5.  (rw * (use arl mode: exact))
    ;MKLSET(RANGE(ALIST))⊂(λX.ATOM X)

6.  (rw * (open inclusion mklset))
    ;∀XV.MEMBER(XV,RANGE(ALIST))⊃ATOM XV

7.  (ue ((phi1.|λx.atom x|)(u.|range alist|))
       allp,introduction * )
    ;ALLP(λX.ATOM X,RANGE(ALIST))

8.  (ci arl)
    ;MKLSET(DOM(ALIST))=MKLSET(RANGE(ALIST))⊃ALLP(λX.ATOM X,RANGE(ALIST))
    (label atomrange) ■
```

Now we can prove

Theorem 3. (i) (*Permutp Invalist*)

```
∀ALIST.PERMUTP(ALIST)⊃PERMUTP(INVALIST(ALIST))
```

Proof. By our application of the Pigeon Hole Principle we know that

```
∀ALIST.PERMUTP(ALIST)⊃INJECTP(ALIST)
```

(see *Permutp Injectp*, Section 4.3.5). Lines 3 and 4 will give permutp(invalist (alist)) (line 10), using the fact that

(i) dom(invalist(alist)) is range(alist) (line 6) and

(ii) range(invalist(alist)) is dom(alist) (line 7);

these are true because of our Lemma *Atomrange* (line 5).

```
        (proof permutp-invalist)

    1.  (assume |permutp alist |)
        (label pivl)

    2.  (derive |injectp alist|(permutp_injectp pivl))
        ;deps: (PIV1)

    3.  (rw * (open injectp))
        ;FUNCTP(ALIST)∧UNIQUENESS(RANGE(ALIST))
        (label piv2)

    4.  (rw piv1 (open permutp))
        ;FUNCTP(ALIST)∧MKLSET(DOM(ALIST))=MKLSET(RANGE(ALIST))
        (label piv3)

    5.  (derive |allp(λx.atom x,range alist)| (atomrange *))
        (label piv4)

    6.  (derive |dom invalist(alist)=range alist| (dom,invalist *))
        (label piv5)

    7.  (derive |range invalist(alist)=dom alist| (range-invalist piv4))
        (label piv6)

    8.  (trw |uniqueness dom(invalist(alist))| piv2 (use piv5))
        ;UNIQUENESS(DOM(INVALIST(ALIST)))
        (label piv7)

    9.  (trw |mklset dom(invalist(alist))=mklset range(invalist(alist))|
           piv3 (use piv5 piv6))
        ;MKLSET(DOM(INVALIST(ALIST)))=MKLSET(RANGE(INVALIST(ALIST)))
        (label piv8)

    10. (trw |permutp invalist(alist)| piv7 piv8
          (open permutp functp) (use invalistsort piv4 mode: exact))
        ;PERMUTP(INVALIST(ALIST))
        ;deps: (PIV1)

    11. (ci piv1)
        ;PERMUTP(ALIST)⊃PERMUTP(INVALIST(ALIST))
        (label permutp-invalist) ■
```

6. Representation using Lists of Numbers.

The rest of this paper contains the proof that permutations (of a. finite set) form a. group, with functions being represented by lists of numbers,

6.1. General Comments on the Choice of the LISP Functions or Predicates.

After the main features of a certain representation have been chosen, many variations are possible, not, a.11 equally desirable. Our representation of permutations by lists of numbers "lists the range" in the order given by the domain. However, given a 1-1 finite function $h : \mathbf{N}_n \to \mathbf{N}$, we could have "listed the domain" in the order given by the range †. We could represent the operation of "applying a list v" to a number k in the domain of h by `(λn.position(v,n))(k)` where position gives the number corresponding to the (first) position of the number n in the list u. Then we have

$$\texttt{position}(\texttt{v},k) = h(k).$$

Our representation has the advantage that it allows the representation of any finite function. not only injections and permutations.

Given a certain LISP program, different formal representation are possible. For instance, when we express our programs in the language of EKL, we can either represent them as functions or as predicates. Logically speaking. the representations are equivalent, but one should not expect the matter to be irrelevant for automatic proof checking. There are many programs computing the same function and many properties can be used to characterize them.

Sometimes it seems quite clear what we need: for instance, the operation of composition of two functions. represented as by a LISP function, should be

```
(define compose |∀u v.u⊗v=mapcar(λi.appl(u,i),v)|)
(label composedef )
```

or, avoiding `mapcar`,

```
(define compose |∀u v x.(u⊗nil)=nil∧
                        (u⊗(x.v))=(nth(u,x)).(u⊗v)|
        listinductiondef)
(label composedef)
```

Given our definition of perm, the identity function for permutations of n elements is the list, `(1 ...* n)`. The most obvious recursive programs generating it are represented either by

```
(decl (indent) (type: |ground→ground|))
(define ident |∀n.ident(0)=nil∧
              ident(n')=ident(n)*list(n')| inductive-definition)
```

where `*` is the LISP function append. or by

† This representation is practical only if h is indeed a permutation. If the range is not a segment of N , we would need some place-holder to mark the places not in the range of h — and. of course. this representation doesn't make sense if h is not 1-1.

```
(define identl |∀n.ident1(0)=nil∧
                  ident1(n')=n'.ident1(n)I inductive-definition)
(define ident |∀n.ident(n)=reverse(ident1(n)) I).
```

These definitions. however, need not be the best choice from the point of view of automatic proof checking. For we will try to construct. proofs by induction on numbers, or lists, or both. The first definition would be all right, except that lists are defiued recursively using cons, not append, so one would need some extra lemmata about append. The second definition is perhaps worse, because of the use of reverse. One can use the standard trick of introducing an auxiliary function with an extra parameter:

```
(defax identl |∀x u n i.ident1(i,0)=nil∧
                        ident1(i,n')=i.ident1(i',n) I)
(label identdef 1)

(define ident |∀n.ident(n)=ident1(0,n) I)
(label identdef) ·
```

If we want to introduce identity by a. predicate, we may be tempted to follow the above definition:

```
(decl (identpl) (type: |ground⊗ground⊗ground→truthval|))

(defax identpl |∀x u n i.identp1(nil,i,n)∧identp1(u,i,0)∧
                         identp1(x.u,i,n')=(x=i∧identp1(u,i',n))|)
(label identdef 1)

(decl (identp) (type: |ground→truthval|))
(define identp IVu.identp(u)=identpl(u,O,length(u)) I)
(label identdef)
```

The definition of identpl is by double recursion on numbers and lists. This complicates the subsequent inductions.

The LISP function inverse is defined using f stposition by recursion on numbers:

```
(def ax invers 1
  |∀u i n.invers1(u,i,0)=nil∧invers1(nil,i,n)=nil∧
         invers1(u,i,n')=if null(fstposition(u,i))
                             then nil
                             else fstposition(u,i) .invers1(u,i',n) I)
(label inversdef 1)

(define inverse|∀u.inverse(u)=invers1(u,0,length(u))|)
(label inversdef )
```

One could represent it by the predicate:

```
(def ax inverspl
 |∀u v i.(inversp1(nil,v,i)≡null(v)∨null(fstposition(v,i)))∧
        (inversp1(x.u,v,i)≡x=fstposition(v,i)∧inversp1(u,v,i'))|)
(label inversdef 1)

(define inverse|∀u.inversp(u,v)=inversp1(u,v,0)|)
(label inversdef ) .
```

Notice that `inversp1` is defined by recursion on lists. The choice of the definition of a function determines the form of induction to be used in the proofs. (For a systematic use of this remark, see Boyer and Moore [1979].) However, the principle of proof is by no means *uniquely* determined in this way. For instance two objects with different recursive definitions in the same statement may already produce a. puzzle. Often to find the right form of induction is a. nontrivial contribution required from human interaction. Consider one of our main facts, say the theorem *Right Inverse*:

```
∀u.perm(u)⊃u⊛inverse(u)=ident(length(u))
```

If we have defined our operations as functions. the function compose suggests that we try an induction on lists. or maybe a double induction on lists and numbers, since inversl and `ident` 1 are defined by recursion on numbers. This cannot work: `perm(x .u)` does not imply `perm(u)`.

We need to use induction *locally*, with respect to a. single list. We must assume that a list is a. permutation and prove facts about it by scanning it, without assuming that its sublists are permutations. *Nthcdr Induction* is such a local form of induction:

```
∀phi u.phi(nil)∧(∀n.n<length(u)⊃(phi(nthcdr(u,n'))⊃
      phi(nth(u,n).nthcdr(u,n'))))⊃phi(u)
```

Yet this doesn't solve our problem. Since inverse and `ident` are defined by recursion on numbers, a promising route is to use induction on *numbers* instead of *lists* and expand the definitions of inverse and `ident`.

Certainly our search for a. proof strategy is not a blind process by trial and errors. guided by some hints from the definition of the objects. We have a purpose and an intuitive idea. namely to construct the identity list using the fact that for a.11 n less than the length of u

```
nth(u,fstposition(u,n))=n.
```

Indeed we have defined inverse through f stposition in order to do this. What we are searching is a strategy of proof that accomplishes it. Iii our search now we know that we cannot use induction on the given list u. To use induction on n seems intuitively right: our theorem asserts that two mathematical objects behave like the identity function, i.e. given a number they return the same number. Our proof should be based upon this property of the identity function.

We are satisfied with such intuitive guidelines in interactive proof checking. For the issue here is still the adequacy of representation, and not the automatic heuristics of mathematical proofs. Guided. say, by the above remarks, we will attempt to prove by induction on n

```
perm(u)∧m<length(u)⊃(u⊛invers1(u,m,n))=ident1(m,n).
```

and we ask whether this is the best strategy for an effective mechanical simulation of the intuitive proof.

Further remarks may give suggestions for the kind of improvement we are interested in. namely the improvement of the efficiency of the entire proof.

Since we want. to see that two lists are equal, it is natural to use the lemma. *Extensionality* for lists:

```
length u=length v⊃((∀i.i<length u⊃nth(u,i)=nth(v,i))⊃u=v).
```

This suggests that we break the proof in two parts, proving first something about lengths and second something about *nth* elements. Indeed, the function nth may be the *linking notion* that allows to choose induction on numbers rather than on lists as the basic proof strategy.

If we choose to represent our operation by predicates, it is not, convenient to use the predicates identpl and inverspl (although they are close to the recursive definition of our programs). It is better to define the predicate 'u is the identity' as

```
(defax id |∀u.id(u)≡(∀n.n<length u⊃nth(u,n)=n)|)
(label id-def )
```

and 'u is the inverse of v' as

```
(defax inv |∀u v.inv(u,v)≡(∀n.n<length u⊃nth(u,n)=fstposition(v,n))|
(label invdef )
```

Then 'u is the composition of v and w' becomes:

```
(define comp
    |∀u v w.comp(u,v,w)≡length u=length w∧
                        (∀n.n<length u⊃nth(u,n)=nth(v,nth(w,n)))|>
(label compdef )
```

The proof of our theorem as

```
∀u v w.perm(w)∧inv(u,w)∧comp(v,w,u)∧length u=length w⊃id(v)
```

then follows simply by expanding the definitions. without a need of complicated inductions.

We consider these definitions as more 'abstract and 'extensional': the 'intensional' features of the programs computing our functions are abstracted away. Since nth is the function that interprets our notion of application, they describe the properties of applications, together with the properties of the lists used in our representation.

Therefore. when dealing with functions instead of predicates, it is convenient, to prove these definitions as basic properties of our functions, rather than carrying through the proofs directly. We follow this strategy in the following proofs. In the representation through fu nct ions we will prove the lemmata *Nth Compose, Main Id, Main Inv,* showing that our functions compose. ident and inverse have the properties described by the predicates comp, id and inv. respectively.

On the other hand, to represent the operations as primitive recursive fnnctionals and to prove facts by the corresponding induction is in many cases the natural choice: very often. representing the operations by functions, rather than by predicates, allows for simpler proofs. (An clear example is the proof that composition of permutations is associative).

In conclusion. inspection of our proof should be convincing evidence that. the following st rategy is the most. efficient in terms of the overall organization of the material.

1) It is convenient to use the function compose to represent composition of functions. for the proof of associativity is much shorter;

2) It is convenient to characterize the identity permutation by the predicate id. and the predicate inv for the operation of inversion of permutations. We obtain elegant proofs of the properties of t lie left and right identity and of left and right. inverse.

3) Finally. we can easily prove that

```
∀n.id(ident n)
```

and

```
∀u.perm(u)⊃inv(inverse u,u)
```

Using these facts. we can derive theorems 2 and 3 for the specific functions `ident` and inverse as corollaries.

However, in order to give the most convincing evidence for the gain in efficiency obtained in this way, we will consider most of the proofs in the two formulations. Occasionally we will show also how a. direct proof looks like in the representation using functions.

6.2. Definitions of Composition, Identity, Inverse.

6.2.1. Functions as Lists: Using Predicates.

```
;definitions of composition,identity, inverse as predicates
(proof comp_pred)

;composition of functions

1  (decl (comp) (type: |ground⊗ground⊗ground→truthval|) (syntype: constant)
         (bindingpower: 930))

2  (define comp
       |∀u v w.comp(u,v,w)≡length u=length w∧
                        (∀n.n<length u⊃nth(u,n)=nth(v,nth(w,n)))|)
   (label compdef )

   ;the identity function

3  (decl (id) (type: |ground→truthval|))
4  (defax id |∀u.id(u)≡(∀n.n<length u⊃nth(u,n)=n)|)
   (label id-def )

   ;the inverse of a function

5  (decl (inv) (type: |ground⊗ground→truthval|))
6  (defax inv |∀u v.inv(u,v)≡(∀n.n<length u⊃nth(u,n)=fstposition(v,n))|)
   (label invdef )
```

Remark. Using list representation for functions the assumption that the functions are defined on the same domain is represented by the condition that our lists have the same length. In our situation we consider permutations of a. finite set. We assume that the lists are of a fixed length. We characterize u as the composition of v and w by the property

```
(∀n.n<length u⊃nth(u,n)=nth(v,nth(w,n))).
```

In order to speak of *the* composition of u and w we have to add the condition that,

```
length(u)=length(w).
```

Similarly for the inverse of a function.

6.2.2. Functions as Lists: Using Functions.

If every element of u is a number less then length(v), then it makes sense to apply v to each element of u (since we defined appl (v, x) to be nth(v, x)). In this case we may say that v is defined as a function over the domain u.

```
(proof camp,fnct)

1.  (decl def_appl (type: | @u⊗@u→truthval| ) )
2.  (define def,appl |∀u v.def_appl(v,u)≡allp(λx.natnum(x)∧x<length(v),u)|)
    (label def_appl_fact)
```

Composition of functions:

```
3.  (decl (compose) (infixname: |⊗|) (type: |ground⊗ground→ground|)
    (syntype: constant)(bindingpower: 930))

4.  (define compose |∀u v x.(u⊗nil)=nil∧
                             (u⊗(x.v))=(nth(u,x)).(u⊗v)| listinductiondef)
    (label composedef)
```

The identity function:

```
5.  (decl (identl) (type: |ground⊗ground→ground|))
6.  (defax identl |∀x u n i.ident1(i,0)=nil∧
                             ident1(i,n')=i.ident1(i',n) I)
    (label identdef 1)

7.  (decl (ident) (type: |ground→ground|))
8.  (define ident |∀n.ident(n)=ident1(0,n) I)
    (label identdef)
```

The inverse of a function:

```
9.  (decl (inversl) (type: |ground⊗ground⊗ground→ground|))
10. (defax inversl
      |∀u i n.invers1(u,i,0)=nil∧invers1(nil,i,n)=nil∧
              invers1(u,i,n')=if null(fstposition(u,i))
                                 then nil
                                 else fstposition(u,i) .invers1(u,i',n) I)
    (label inversdef 1)

11. (decl (inverse) (type: |ground→ground|))
12. (define inverse|∀u.inverse(u)=invers1(u,0,length(u))|)
    (label inversdef)
```

6.3. Preliminaries.

We collect here facts about definiteness, sort and length of the lists resulting from our operat ions in the representation using functions. We prove facts about concrete LISP programs that perform the operations 'composing' two lists and taking the 'inverse' of a list: hence we obtain more information t han in the proofs using predicates.

Remark. The proofs of in the representation by functions require tedious comput at ions involving the function minus. Typically, in a proof by induction on n. for `n<length` u. the induction step cont ains an espression like

```
LENGTH (INVERS1(U,(LENGTH U-N')',N))=N))
```

to be simplified as

```
LENGTH (INVERS1(U,LENGTH U-N,N))=N))
```

Such replacement is dependent on the truth of the clause `N<LENGTH U`.

We have collected in the proof *MINUS* a sequence of facts of the form

```
;labels: MINUSFACT10
∀N M.N<M⊃M-N=(M-N')'
```

to be used as rewriters in similar cases. In fortunate situations the truth of the condition is immediately recognized by the decision procedure. Often a derivation is needed from other facts. e.g. from

```
;labels: LESS,LESSEQSUCC
∀M N.M<N=M'≤N
```

and we may need to do the substitution 'by brute force'. with some tedium and pain.

6.3.1. Preliminaries: Condition for Definiteness and Sorts of the Functions.

The condition for v to be applicable to u as domain is formulated recursively in *Def Appl Fact*. Now we give a sufficient condition for *Def Appl Fact*.

```
1.  (assume lint0 u|)

2.  (assume |length u≤length v|)

3.  (rw -2 (open into))
    ;∀N.N<LENGTH U⊃NATNUM(NTH(U,N))∧NTH(U,N)<LENGTH U

4.  (trw |∀n.n<length u⊃natnum nth(u,n)∧nth(u,n)<length v| *
        (less,lesseq-fact1 -2))
    ;∀N.N<LENGTH U⊃NATNUM(NTH(U,N))∧NTH(U,N)<LENGTH V

5.  (ue ((phi1.|λx.natnum x∧x<length v|)(u.u)) nth,allp * )
    ;ALLP(λX.NATNUM(X)∧X<LENGTH V,U)

6.  (trw |def_appl(v,u)| (open def_appl) * )
```

```
;DEF_APPL(V,U)

7.  (ci (-6 -5))
    ;INTO(U)∧LENGTH U≤LENGTH V⊃DEF_APPL(V,U)
    (label def _appl_condition)   ■
```

In particular. the same condition holds for permutations of the appropriate length.

```
a.  (rw def_appl_condition (open lesseq) (use normal mode: always))
    ;∀U V.(INTO(U)∧LENGTH U=LENGTH V⊃DEF_APPL(V,U))∧
    ;     (INTO(U)∧LENGTH U<LENGTH V⊃DEF_APPL(V,U))

9.  (derive |perm u∧length u = length v⊃def_appl(v,u)|
 (def_appl_condition *)(open perm onto))
    (label def _appl_condition1)   ■
```

We prove that the results of our operations have the right sorts.

compose:

```
10. (ue (phi |λu.def_appl(v,u)⊃listp v⊗u|) listinduction
        (part 1 (open def_appl allp compose )))
    ;∀U.DEF_APPL(V,U)⊃LISTP V⊗U
    (label sortcomp) (label simpinfo)   ■
```

ident:

```
11. (ue (a |λn.∀m.listp ident1(m,n)|) proof-by-induction
           (open identl))
    ;∀N M.LISTP IDENT1(M,N)
    (label ident_sort1) (label simpinfo)   ■

12. (trw |∀n.listp ident(n)|(open ident) * )
    ;∀N.LISTP IDENT(N)
    (label ident_sort) (label simpinfo)   ■
```

inverse:

```
13. (ue (a |λn.∀i.listp invers1(u,i,n)|) proof-by-induction
             (open inversl) posfacts)
    ;∀N I.LISTP INVERS1(U,I,N)
    (label invers_sort1) (label simpinfo)   ■

14. (trw |listp inverse(u)| (open inverse) * )
    ;LISTP INVERSE(U)
    (label inverse-sort) (label simpinfo)   ■
```

6.3.2. Preliminaries: Length of Compose.

Lemma 6.1. (*Length Compose*)

```
∀U W.DEF_APPL(W,U)⊃LENGTH (W⊗U)=LENGTH(U)
```

Proof. By *Nthcdr Induction* applied to u. For u = NIL, the result is given by the definition of compose. Assume that

```
length(w ⊗ nthcdr(u,n'))=length(nthcdr(u,n')),
```

for n' less than length(u). We would like to have:

```
length(w ⊗ (nth(u,n).nthcdr(u,n'))=length(nth(u,n).nthcdr(u,n')).
```

Since nth(u,n) is always an S-expression, we can apply the definition of compose: the left hand side becomes

```
length(nth(w,nth(u,n)).(w⊗nthcdr(u,n'))).
```

The inductive step will be proved if we show that the terms have proper sorts, under the assumption of line 1. Lines 3 – 9 do this.

```
     (proof length-compose)

1.   (assume |def_appl(w,u)|)
     (label l_c_1)

2.   (rw * (open def,appl))
     (label l_c_2)
     ;ALLP(λX.NATNUM(X)∧X<LENGTH W,U)
```

Since w is defined as an application on u as domain (line 1), nth(u,n) is a natural number less than length(w). Therefore nth(w,nth(u,n)) is an S-expression (line 5).

```
3.   (assume |n<length(u)|))
     (label l_c_3)

4.   (ue ((u.u)(x.|nth(u,n)|)(phi1.|λx.natnum(x)∧x<length(w)|>>
         allp_elimination
         nthmember sexp,nth l_c_3 l_c_2)
     ;NATNUM(NTH(U,N))∧NTH(U,N)<LENGTH W
     (label l_c_4)

5.   (trw |sexp(nth(w,nth(u,n)))| sexp_nth l_c_4)
     ;SEXP NTH(W,NTH(U,N))
     (label l_c_sort1)

6.   (ci l_c_3)
     ;N<LENGTH U⊃SEXP NTH(W,NTH(U,N))
     (label l_c_7)
```

Moreover, w is defined as an application on any part of u, since it is defined on u (using *Allp Nthcdr*). Therefore, using *Sortcomp*, we see that w ⊗ nthcdr(u,n') is a list.

```
7.  (derive |allp(λx.natnum(x)∧x<length w,nthcdr(u,n'))|
    (allp_nthcdr l_c_2))
    ;ALLP(λX.NATNUM(X)∧X<LENGTH W,NTHCDR(U,N'))

8.  (derive |listp(w⊗nthcdr(u,n'))|(* sortcomp))
    (label l_c_sort2)

9.  (ci l_c_3)
    ;N<LENGTH U⊃LISTP W⊗NTHCDR(U,N')
    (label l_c_8)
```

The result follows by *Nthcdr Induction.*

```
10. (ue ((phi.|λu.length(w⊗u)=length(u)|)(u.u))
        nthcdr,induction
        (part 1 (open compose length )) l_c_7 l_c_8)
    ;LENGTH (W⊗U)=LENGTH U

11. (ci l_c_1)
    ;DEF_APPL(W,U)⊃LENGTH (W⊗U)=LENGTH U
    (label length-compose) ■
```

6.3.3. Preliminaries: Length of Ident.

Lemma 6.2. *(Length Ident)*

VU N.LENGTH (IDENT(N))=N

Proof.

```
1.  (ue (a |λn.∀m.length ident1(m,n)=n|)
        proof-by-induction
        (open identl))
    ;∀N M.LENGTH (IDENT1(M,N))=N
    (label length-identl) (label simpinfo)

2.  (trw |∀N.LENGTH (IDENT(N))=N| * (open ident))
    (label length-ident) (label simpinfo)    ■
```

6.3.4. Preliminaries: Length of Inverse.

Lemma *6.3. (Lengthinverse*

```
∀U.PERM(U)⊃LENGTH (INVERSE(U))=LENGTH u
```

Remark. Example 8. It may be instructive to consider the heuristics of the proof. The first problem is to find the appropriate sublemma. inverse is defined in terms of the auxiliary function inversl and the latter is defined by recursion on its third argument:

```
;labels: INVERSDEF1
   VU I N.INVERS1(U,I,0)=NIL∧INVERS1(NIL,I,N)=NIL∧
          INVERS1(U,I,N')=
          (IF NULL FSTPOSITION(U,I)
              THEN NIL
              ELSE FSTPOSITION(U,I).INVERS1(U,I',N))
```

Hence it is reasonable to try a proof by induction on n. The reader should see why it is not reasonable to try induction on u. We assume perm u and try to prove

```
(∀N.N≤LENGTH U⊃LENGTH (INVERS1(U,LENGTH U-N,N))=N)
```

At this point we immediately see that
(i) the base case follows by expanding the definitions:
(ii) in the inductive step f stposition(u,length u-n) will not be null since

length u-n'<length u and onto(u):

(iii) to apply the induction hypothesis in the inductive step we need the lemma

```
;labels: SUCC,LESSEQ,LESSEQ
∀M N.M'≤N⊃M≤N
```

We can ask EKL to prove it and see precisely in what form the above information must be presented to EKL or what other facts we may have overlooked.

```
(ue (a |λn.n≤length u⊃length inversl(u,length u-n,n)=n|)
    proof-by-induction
   (open inversl) succ_lesseq_lesseq)
;(∀N.(N≤LENGTH U⊃LENGTH (INVERS1(U,LENGTH U-N,N))=N)⊃
;    (N'≤LENGTH U⊃
;     ¬NULL FSTPOSITION(U,LENGTH U-N')∧
;     LENGTH (INVERS1(U,(LENGTH U-N')',N))=N))⊃
;(∀N.N≤LENGTH U⊃LENGTH (INVERS1(U,LENGTH U-N,N))=N)
```

This test informs us that EKL has done the base case as expected and has expanded the definition of inversl in the induction step. In both cases of'the conditional definition of inversl. the definition of length has been expanded as desired. giving 0=n' if

(⋆) NULL FSTPOSITION(U,LENGTH U-N')

and

```
(★★)                LENGTH(INVERS1(U,(LENGTH U-N')'))=N
```

otherwise: so the clause of the form if p then false else q has become ¬p ∧ q.

Now we are confident that EKL will prove the negation of (★), according to the argument given in (ii) above, with the information contained in *Posfacts* and *Minusfact11*:

```
;labels:SIMPINFO POSFACTS
 ∀U Y.(NULL FSTPOSITION(U,Y)⊃¬MEMBER(Y,U))∧
      (MEMBER(Y,U)⊃NATNUM(FSTPOSITION(U,Y)))∧
      (NULL FSTPOSITION(U,Y)∨NATNUM(FSTPOSITION(U,Y)))

;labels: MINUSFACT11
∀N M.M<N⊃N-M'<N
```

and that EKL will see that the induction hypothesis implies (★★), if we add

```
;labels: MINUSFACT10
∀N M.N<M⊃M-N=(M-N')'
```

In both cases we need also

```
;labels: LESS,LESSEQSUCC
;∀M N.M<N≡M'≤N
```

etc. The details of the proof follow. □

Proof.

```
1.  (assume |perm(u)|)
    (label li1)

2.  (rw li1 (open perm onto into))
    ;(∀N.N<LENGTH U⊃NATNUM(NTH(U,N))∧NTH(U,N)<LENGTH U)∧
    ;(∀N.N<LENGTH U⊃MEMBER(N,U))
    (label li2)

3.  (ue ((u.|u|) (y.|n|)) posfacts)
    ;(NULL FSTPOSITION(U,N)⊃¬MEMBER(N,U))∧
    ;(MEMBER(N,U)⊃NATNUM(FSTPOSITION(U,N)))

4.  (derive |n<length u⊃¬null fstposition(u,n)| (3 li2))
    (label li3)

5.  (ue ((m.|n|) (n.|length u|)) minusfact11
        (part 1 (use less,lesseqsucc mode: exact)))
    ;N'≤LENGTH U⊃LENGTH U-N'<LENGTH U

6.  (derive |n'≤length u⊃¬null fstposition(u,length u-n')| (5 li3))
    (label li4)

7.  (trw |n'≤length u⊃(length u-n')'=length u-n|
         (use minusfact10)
         (use less-lesseqsucc mode: exact direction: reverse))
    ;N'≤LENGTH U⊃(LENGTH U-N')'=LENGTH U-N

8.  (ue (a |λn.n≤length u⊃length (inversl(u,length u-n,n))=n|)
       proof-by-induction
       (open inversl) (use succ_lesseq_lesseq) (use 7) (use li4))
    ;∀N.N≤LENGTH U⊃LENGTH (INVERSl(U,LENGTH U-N,N))=N

9.  (ue (n |length u|) * (open lesseq))
    ;LENGTH (INVERS1(U,0,LENGTH U))=LENGTH u

10. (trw |length inverse(u)=length u| (open inverse) *
    ;LENGTH (INVERSE(U))=LENGTH U
    ;deps: (LI1)

11. (ci li1)
    ;PERM(U)⊃LENGTH (INVERSE(U))=LENGTH u
    (label lengthinverse) ∎
```

6.4. Theorem 1: Composition of Permutations.

6.4.1. Using predicates: Composition of Permutations is a Permutation.

We prove the following theorem:

Theorem 1 *(Composition)*

(i) ∀U V W.PERM(V)∧PERM(W)∧LENGTH V=LENGTH W∧COMP(U,V,W)⊃PERM(U)
(ii) ∀U V W.COMP(U,V,W)∧COMP(U1,V,W)⊃U=U1

The proof of (i) is long but not hard.

Proof. Assume that v and w are permutations (lines 1 and 2), have the same length (line 3) and u is the result of 'composing' v and w (line 4). We show that

(i) u is into (line 17) and
(ii) u is onto (line 32).

(i), is a matter of expanding definitions. If m is less than the length of u (and so of w and of v), then nth(w,m) is a natural number less than the length of w (line 10) and nth(v,nth(w,m)) is a natural number less than the length of v (line 11). But this is just nth(u,m), by the definition of composition (line 15). 'Intoness' follows.

```
    ;composition of permutation is a permutation
    (proof comp_perm)

1.  (assume |perm(v)|)
    (label cp_pm1)

2.  (assume |perm(w)|)
    (label cp_pm2)

3.  (assume |length v=length w|)
    (label cp_pm3)

4.  (assume |comp(u,v,w)|)
    (label cp_pm4)
```

Rewrite:

```
5.  (rw cp_pm1 (open perm into onto))
    (label cp_pm5)
    ;(∀N.N<LENGTH V⊃NATNUM(NTH(V,N))∧NTH(V,N)<LENGTH V)∧
    ;(∀N.N<LENGTH V⊃MEMBER(N,V))

6.  (rw cp_pm2 (open perm into onto))
    (label cp_pm6)
    ;(∀N.N<LENGTH W⊃NATNUM(NTH(W,N))∧NTH(W,N)<LENGTH W)∧
    ;(∀N.N<LENGTH W⊃MEMBER(N,W))

7.  (rw cp_pm4 (open comp ))
    (label cp_pm7)
    ;LENGTH U=LENGTH W∧(∀N.N<LENGTH U⊃NTH(U,N)=APPL(V,NTH(W,N)))
```

A straightforward verification.

```
8.   (assume |m<length(u) |)
     (label cp_pm8)

9.   (rw * (use cp_pm7 mode: always))
     (label cp_pm9)
     ;M<LENGTH W

PO. (derive |natnum(nth(w,m))∧nth(w,m)<length v| (cp_pm6 *)
            (use cp_pm3 mode: exact))
```

So we can obtain the desired result...

```
11. (trw |NATNUM(NTH(V,NTH(W,M))∧NTH(V,NTH(W,M))<LENGTH V| (* cp_pm5))
    (label cp_pm10)
```

... and open appl in line 7

```
12. (derive |nth(u,m)=nth(v,nth(w,m))| (cp_pm7 cp_pm8)
            (open appl)  (use -2))

13. (rw cp_pm10 (use * mode: exact direction: reverse))
    ;NATNUM(NTH(U,M))∧NTH(U,M)<LENGTH V
    (label cp_pm11)

14. (trw |length u=length v| (use cp_pm7 cp_pm3 mode: always))
    ;LENGTH U=LENGTH V

15. (rw cp,pm11  (use * mode: exact direction: reverse))
    ;NATNUM(NTH(U,M))∧NTH(U,M)<LENGTH U
    ;deps: (CP,PM1 CP_PM2 CP_PM3 CP_PM4 CP_PM8)

16. (ci cp_pm8)
    ;M<LENGTH U⊃NATNUM(NTH(U,M))∧NTH(U,M)<LENGTH U

17. (trw |into u| (open into) * )
    (label cp_into)
    ;INTO(U)
    ;deps: (CP_PM1 CP_PM2 CP_PM3 CP_PM4)
```

The second part, the proof of (ii), is sligthly more complicated. Any m less than the length of **u** (and so of v and of w) is a member of v (line 19), since v is onto.

```
18. (rw cp_pm9 (use cp_pm3 mode: exact direction: reverse))
    ;M<LENGTH V

19. (trw |member(m,v)| (* cp_pm5))
    ;MEMBER(M,V)
    (label cp_pm20)
```

Therefore we can find a number jv less than the length of **v** such that m is the jv-th element of **v** (line 21).

```
20. (derive |∃j.j<length(v)∧nth(v,j)=m| (* member-nth))
    (label cp_pm21)
    ;deps: (CP_PM1 CP_PM3 CP_PM4 CP_PM8)

21. (define jv |jv<length(v)∧nth(v,jv)=m| * )
    (label cp_pm22)
```

Again, since w is onto, jv is a member of w (line '23).

```
22. (rw * (use cp_pm3 mode: exact))
    ;JV<LENGTH W∧NTH(V,JV)=M

23. (trw |member(jv,w)| (* cp_pm6))
    ;MEMBER(JV,W)
```

And again we can find a number kv less than the length of w such that jv is the kv-th element of w (line '25).

```
24. (derive |∃k.k<length(w)∧nth(w,k)=jv| (* member-nth))
    ;deps: (CP_PM1 CP_PM2 CP_PM3 CP_PM4 CP_PM8)

25. (define kv |kv<length(w)∧nth(w,kv)=jv| * )
    (label cp_pm23)
```

So m is nth(v,nth(w,kv)) (line 24); but this is just nth(u,kv) (line SO), by the definition of composition.

```
26. (rw cp_pm22 (use * mode: always direction: reverse))
    ;NTH(W,KV)<LENGTH V∧NTH(V,NTH(W,KV))=M
    (label cp_pm24)

27. (trw |kv<length(u)| cp_pm23 (use cp_pm7 mode: always))
    ;KV<LENGTH U
    (label cp_pm25)

28. (trw |natnum nth(w,kv)| cp_pm23)
    ;NATNUM(NTH(W,KV))

29. (derive |nth(u,kv)=nth(v,nth(w,kv))| (cp_pm7 cp_pm25)
       (open appl) (use *) )

30. (rw * (use cp_pm24 mode: always))
    ;NTH(U,KV)=M
```

The last equation allows us to apply lemma *Nthmember* and conclude that m is a member of u.

```
31. (derive |member(m,u)| nthmember
       cp_pm25  (use * mode: exact direction: reverse>)
    ;deps: (CP,PM1 CP_PM2 CP_PM3 CP_PM4 CP_PM8)

32. (ci cp_pm8)
    ;M<LENGTH U⊃MEMBER(M,U)
    (label cp,onto)

33. (trw |perm u| (open perm onto) cp,into cp,onto)
    ;PERM(U)
    ;deps: (CP_PM1 CP_PM2 CP_PM3 CP_PM4)

34. (ci (cp_pm1 cp_pm2 cp_pm3 cp_pm4))
    ;PERM(V)∧PERM(W)∧LENGTH V=LENGTH W∧COMP(U,V,W)⊃PERM(U)
    (label perm_composition) ■
```

Composition of functions is unique:

```
35. (trw |comp(u,v,w)∧comp(u1,v,w)⊃u=u1| (open comp) extensionality)
    ;COMP(U,V,W)∧COMP(U1,V,W)⊃U=U1
    (label comp_uniqueness) ■
```

6.4.2. Using Predicates: Composition is Associative.

Finally we prove associativity:

Theorem 1 (iii) (*Associativity Pred*)

```
∀U U1 V V1 W1 W2 W3.INTO(W3)∧LENGTH W2=LENGTH W3∧
                     COMP(V,W1,W2)∧COMP(U,V,W3)∧
                     COMP(V1,W2,W3)∧COMP(U1,W1,V1)⊃U=U1
```

Proof. The aim is to apply extensionality. In view of an application of *Extensionality* (line 26), we want to prove that for all $n <$ length(u)

$$\text{nth(u,n)} = \text{nth(u1,n)}.$$

The facts needed follow from the definitions. However, a lot of rewriting is required not only to expand definitions, but also to find the right matching (e.g. see the derivation of line 16 from lines 10 and 1.5). The decision procedure is often applied, since the definition contains a. conditional clause. More specifically. we have to perform the following substitutions:

nth(u,n)	= nth(v,nth(w3,n))	(line 10)	if n<length(u)
	= nth(w1,nth(w2,nth(w3,n)))	(line 16)	if natnum(nth,w3) and nth(w3,n)<length(v)
	= nth(w1,nth(v1,n)	(from 16, 18)	if n<length(v1)
	= nth(u1,n)	(line 23)	if n<length(u1).

It would take a. lot of work to mechanically perform all the matching involved iii these steps. if all possible substitutions had to be attempted at random. It is reasonable to expect human guidance of the proof checker. Therefore one cannot expect a. proof in few lines.

This is in sharp contrast to the proof using functions, consisting of few straightforward inductions on lists and numbers.

```
     (proof comp_associative)

1.   (assume |into(w3)|)
     (label ca1)

2.   (assume |length w2=length w3|)
     (label ca2)

3.   (assume |comp(v,w1,w2)|)
     (label ca3)

4.   (assume |comp(u,v,w3)|)
     (label ca4)

5.   (assume |comp(v1,w2,w3)|)
     (label ca5)

6.   (assume |comp(u1,w1,v1)|)
     (label ca6)

7.   (assume |n<length u|)
     (label ca7)

8.   (rw ca4 (open comp))
     ;LENGTH U=LENGTH W3∧(∀N.N<LENGTH U⊃NTH(U,N)=NTH(V,NTH(W3,N)))
     (label ca8)
     ;deps: (CA4)

9.   (derive |n<length(w3)| (ca7 ca8))
     (label ca9)
     ;deps: (CA4 CA7)

10.  (derive |nth(u,n)=nth(v,nth(w3,n))| (ca7 ca8))
     (label ca10)
     ;deps: (CA4 CA7)

11.  (rw ca1 (open into))
     ;∀N.N<LENGTH W3⊃NATNUM(NTH(W3,N))∧NTH(W3,N)<LENGTH W3
     ;deps: (CA1)

12.  (derive |natnum(nth(w3,n))∧nth(w3,n)<length(w2)| (ca9 * ca2))
     (label ca11)
     ;deps: (CA1 CA2 CA4 CA7)

13.  (rw ca3 (open comp))
     ;LENGTH V=LENGTH W2∧(∀N.N<LENGTH V⊃NTH(V,N)=NTH(W1,NTH(W2,N)))
     (label ca12)
```

```
;deps: (CA3)

14. (derive |nth(w3,n)<length(v)| (call ca12))
    (label ca13)
    ;deps: (CA1 CA2 CA3 CA4 CA7)

15. (derive |∀n.n<length(v)⊃nth(v,n)=nth(w1,nth(w2,n))| ca12)
    (ue (n |nth(w3,n) I) * call ca13)
    ;NTH(V,NTH(W3,N))=NTH(W1,NTH(W2,NTH(W3,N)))
    ;deps: (CA1 CA2 CA3 CA4 CA7)

16. (rw ca10 (use * mode: exact))
    ;NTH(U,N)=NTH(W1,NTH(W2,NTH(W3,N)))
    (label ca14)
    ;deps: (CA1 CA2 CA3 CA4 CA7)

17. (rw ca5 (open comp))
    (label ca20)
    ;LENCTH V1=LENGTH W3∧(∀N.N<LENGTH V1⊃NTH(V1,N)=NTH(W2,NTH(W3,N)))

18. (derive |nth(v1,n)=nth(w2,nth(w3,n))| (ca9 ca20))
    (label ca21)
    ;deps: (CA4 CA5 CA7)

19. (rw ca6 (open comp))
    ;LENGTH U1=LENGTH V1∧(∀N.N<LENGTH U1⊃NTH(U1,N)=NTH(W1,NTH(V1,N)))
    (label ca22)
    ;deps: (CA6)

20. (rw ca9 (use ca20 ca22 mode: always direction: reverse))
    ;N<LENGTH U1
    ;deps: (CA4 CA5 CA6 CA7)

21. (derive |nth(u1,n)=nth(w1,nth(v1,n))| (ca22 *))
    ;deps: (CA4 CA5 CA6 CA7)

22. (rw * (use ca21 mode: exact))
    ;NTH(U1,N)=NTH(W1,NTH(W2,NTH(W3,N)))
    (label ca23)
    ;deps: (CA4 CA5 CA6 CA7)

23. (rw ca14 (use ca23 mode: exact direction: reverse))
    ;NTH(U,N)=NTH(U1,N)
    ;deps: (CA1 CA2 CA3 CA4 CA5 CA6 CA7)

24. (ci ca7)
    ;N<LENGTH U⊃NTH(U,N)=NTH(U1,N)
    (label ca24)
    ;deps: (CA1 CA2 CA3 CA4 CA5 CA6)

25. (trw |length u = length u1| (use ca8 ca22 mode: always)
          (use ca20 mode: always direction: reverse))
    ;LENGTHU=LENGTHU1
    ;deps: (CA4 CA5 CA6)
```

```
26. (ue ((u.u)(v.u1)) extensionality ca24 * )
    ;U=U1
    ;deps: (CA1 CA2 CA3 CA4 CA5 CA6)

27. (ci (cal ca2 ca3 ca4 ca5 ca6))
    ;INTO(W3)∧LENGTH W2=LENGTH W3∧
    ;COMP(V,W1,W2)∧COMP(U,V,W3)∧
    ;COMP(V1,W2,W3)∧COMP(U1,W1,V1)⊃U=U1
    (label associativity-pred) ■
```

6.4.3. Using Functions: the Lemma Nth Compose.

We prove first the lemma. *Nth Compose*,i.e. the basic property of composition, that was taken as definition of the predicate comp.

Lemma 6.4. (*Nth* Compose)

VU N.DEF_APPL(V,U)∧N<LENGTH U⊃NTH(V⊗U,N =NTH(V,NTH(U,N))

Proof. By double induction on *n* and II:

```
(proof nth-compose)

;labels: DOUBLEINDUCTION1
;(∀U N X.PHI3(NIL,N)∧PHI3(U,0)∧(PHI3(U,N)⊃PHI3(X.U,N'))) ⊃
;(∀U N.PHI3(U,N))
```

One base-case is proved by listinduction:

```
1.  (ue (phi |λu.¬null(u)∧def_appl(v,u)⊃nth(v⊗u,0)=nth(v,nth(u,O)) |)
        listinduction
        (part 1 (open compose nth def_appl allp)) )
    ;∀U.¬NULL U∧DEF_APPL(V,U)⊃CAR (V⊗U)=NTH(V,CAR U)
    (label a_c_base1)
```

. . .and the other is trivial. So:

```
2.  (ue (phi3 |λu n.def_appl(v,u)∧n<length(u)⊃nth(v⊗u,n)=nth(v,nth(u,n))|)
        doubleinductionl
        (part 1 (open compose def,appl allp)) a_c_base1)
    ;∀U N.DEF_APPL(V,U)∧N<LENGTH U⊃NTH(V⊗U,N)=NTH(V,NTH(U,N))
    (label nth-compose) ■
```

Exercise. Prove Theorem 1 in the representation by functions

VU V.PERM U ∧ PERM V ∧ LENGTH U = LENGTH V ⊃ PERM(U⊗V)

wing directly Theorem 1 (*Composition*)

∀U V W.PERM(V)∧PERM(W)∧LENGTH V=LENGTH W∧COMP(U,V,W)⊃PERM(U)

6.4.4. Using Functions: Theorem 1.

Theorem 1 (i) (*Perm Compose)*

```
∀U V.PERM U ∧ PERM V ∧ LENGTH U = LENGTH V ⊃ PERM(U⊗V)
```

The proof is basically the same as in Section **6.4.1.** It can be found in the Appendix.

Theorem 1 (ii) (*Associa tivity* of Composition)

```
∀U V W.PERM(V)∧PERM(U)∧LENGTH V=LENGTH U∧LENGTH W=LENGTH U⊃
       (W⊗V)⊗U=W⊗(V⊗U)
```

Proof of (ii) By listinduction on u:

```
(proof assoc,compose)

1.  (trw |def_appl(w,v)∧def_appl(v,u)⊃(w⊗v)⊗nil=w⊗(v⊗nil>|
         (open compose) sortcomp)
    (label ass_comp_base)

2,  (ue (phi |λu.def_appl(w,v)∧def_appl(v,u)⊃(w⊗v)⊗u=w⊗(v⊗u)|)
     · listinduction
       (part 1#2 (open compose def,appl allp)) sortcomp ass-camp,base
       (use nth,compose ue: ((v.w)(u.v)) ) )
    ;∀U.DEF_APPL(W,V)∧DEF_APPL(V,U)⊃(W⊗V)⊗U=W⊗(V⊗U)
    (label assoc,camp)
```

In particular, the conditions of definedness are satisfied if u and v are permutations of the same length

```
3.  (trw |∀u v w.perm(v)∧perm(u)∧length v=length u∧length w=length u⊃
    (w⊗v)⊗u=w⊗(v⊗u)| assoc_comp
    (use def_appl_condition1 ue: ((u.u)(v.v)) )
    (use def_appl_condition1 ue: ((u.v)(v.w)) ))
      ;∀U V W.PERM(V)∧PERM(U)∧LENGTH V=LENGTH U∧LENGTH W=LENGTH U⊃
      ;(W⊗V)⊗U=W⊗(V⊗U)
      (label associativity,of -composition)   ■
```

Compare with the corresponding result using predicates (Section 6.4.2). An explanation why this proof is much simpler is: both compose and def -appl have a simple definition by recursion on lists. The lemma and the theorem can be proved by a straightforward double induction on lists and numbers. On the other hand. when composition is defined as a predicate. a lot, of rewriting is required to expand the definitions and to perform the right substitutions, and the decision procedure is often applied to justify conditional rewriting. This cannot be done in a few lines.

6.5. Theorem 2: The Identity Permutation.

6.5.1. Using Predicates.

We have the following theorem about identity:
Theorem 2 (i) (*Id Perm*)

```
∀U.ID(U)⊃PERM(U)
```

Proof. Intoness:

```
    ;id implies perm
    (proof idperm)

1.  (trw |id(u)⊃into(u)| (open id into))
    ;ID(U)⊃INTO(U)
    (label p_i1)
```

Ontoness:

```
2.  (assume lid(u) |)
    (label p_i2)

3.  (rw * (open id))
    ;∀N.N<LENGTH U⊃NTH(U,N)=N
    (label p_i3)

4.  (assume |n<length u|)
    (label p_i4)

5.  (derive |member(nth(u,n),u)| (* nthmember))

6.  (derive |member(n,u)| (* p_i4 p_i3))

7.  (ci p_i4)
    ;N<LENGTH U⊃MEMBER(N,U)

8.  (derive |perm u| (p_i1 p_i2 *) (open perm onto))

9.  (ci p_i2)
    ;ID(U)⊃PERM(U)
    (label id-perm) ∎
```

Right and left identity are also easy consequences of the definitions
Theorem 2 (ii) *(Right Id)*

```
VU VW.ID(U)∧COMP(V,W,U)∧LENGTH W=LENGTH U⊃V=W
```

Proof.

```
     (proof identity-right)
 :
 1.  (assume |id(u) I?
     (label id,rl)

 2.  (assume |comp(v,w,u) |>
     (label id_r2)

 3.  (assume |length w=length ul)
     (label id_r3)

 4.  (rw id_r1 (open id))
     ;∀N.N<LENGTH U⊃NTH(U,N)=N
     (label id_r4)

 5.  (rw id_r2 (open comp))
     ;LENGTH V=LENGTH U∧(∀N.N<LENGTH U⊃NTH(V,N)=NTH(W,NTH(U,N)))
     (label id_r5)

 6.  (rw * (use id_r4 mode: always))
     ;LENGTH V=LENGTH U∧(∀N.N<LENGTH U⊃NTH(V,N)=NTH(W,N))
     (label id_r6)

 7.  (trw |length v=length w| (use id_r3 id_r5 mode: always))
     ;LENGTH V=LENGTH W

 a.  (derive |v=w| (extensionality id_r6 *))

 9.  (ci (id_r1 id_r2 id-r3))
     ;ID(U)∧COMP(V,W,U)∧LENGTHW=LENGTHU⊃V=W
     (label id-right) ■
```

Theorem 2 (iii) (*Left Id*)

```
VU VW.ID(U)∧PERM(W)∧LENGTH W=LENGTH U∧COMP(V,U,W)⊃W=V
```

Proof.

```
     (proof identity-left)

-1.  (assume lid(u) |)
     (label id,11)

 2.  (assume |perm w|)
     (label id-12)

 3.  (assume [length w=length u|)
     (label id,13)

 4.  (assume |comp(v,u,w)|)
     (label id-14)
```

```
5.  (rw id-11 (open id))
    ;∀N.N<LENGTH U⊃NTH(U,N)=N
    (label id,15)

6   (rw id-14 (open comp))
    ;LENGTH V=LENGTH W∧(∀N.N<LENGTH V⊃NTH(V,N)=NTH(U,NTH(W,N)))
    (label id-16)

7   (rw id-12 (open perm onto into))
    ;(∀N.N<LENGTH W⊃NATNUM(NTH(W,N))∧NTH(W,N)<LENGTH W)∧
    ;(∀N.N<LENGTH W⊃MEMBER(N,W))
    (label id-17)

8   (trw |∀m.m<length u⊃natnum(nth(w,m))∧nth(w,m)<length u| id-17
         (use id-13 mode: exact direction: reverse))
    ;∀M.M<LENGTH U⊃NATNUM(NTH(W,M))∧NTH(W,M)<LENGTH U
    (label id_18)
```

We can apply the property of the identity function u

```
9.  (trw |∀m.m<length u⊃nth(u,nth(w,m))=nth(w,m)| id-15 * )
    ;∀M.M<LENGTH U⊃NTH(U,NTH(W,M))=NTH(W,M)
    (label id-19)
```

We will use extensionality

```
10. (assume |m<length v|)
    (label id-110)

11. (trw |m<length u| *
         (use id_13 id-16 mode: exact direction: reverse))
    ;M<LENGTH U
    (label id_111)

12. (derive |nth(u,nth(w,m))=nth(w,m)| (id-19 id-111))
```

We use the fact that v is the composition of u and w

```
13. (derive |nth(v,m)=nth(w,m)| (id-16 id,110)
            (use * mode: exact direction: reverse))

14. (ci id,110)
    ;M<LENGTH V⊃NTH(V,M)=NTH(W,M) .

15. (derive |w=v| (extensionality id_16 *))

16. (ci (id-11 id_12 id_13 id,14))
    ;ID(U)∧PERM(W)∧LENGTH W=LENGTH U∧COMP(V,U,W)⊃W=V
    (label id-left) ■
```

6.5.2. Using Functions: the Lemma Main Id.

The main result about the identity' list is the extensional property that was assumed as definition of id.

Lemma 6.5. (*Main* **Id)**

```
∀N.N<M⊃NTH(IDENT(M),N)=N.
```

Proof. First we show the following fact, *Nthcdr Ident*, by induction on n:

```
∀n.n<m⊃nthcdr(ident(m),n)=ident1(n,m-n)
```

(line 8). The lemma then follows easily.

```
(proof id-main)
;id main

1.  (assume |n<m⊃nthcdr(ident(m),n)=ident1(n,m-n)|)
    (label id,mainl)

2.  (assume |n'<m|)
    (label id_main2)

3.  (derive |nthcdr(ident(m),n)=ident1(n,m-n)|
            (id_main1 id_main2 succ,less-less))
    ;deps: (ID_MAIN1 ID_MAIN2)
```

Now we use *Minusfact10* to expand the definition of identl in the right member of the equality.

```
;labels: MINUSFACT10
;∀N M.N<M⊃M-N=(M-N')'

4.  (rw * (use minusfact10 mode: exact) (open identl)
          (use id_main2 succ_less_less mode: exact))
    ;NTHCDR(IDENT(M),N)=N.IDENT1(N',M-N')
    ;deps: (ID_MAIN1 ID_MAIN2)
```

The inductive step is concluded by the use of *Cdr Nthcdr*

```
;labels: CDR,NTHCDR
;∀U N.CDR NTHCDR(U,N)=NTHCDR(U,N')

5.  (trw |nthcdr(ident m,n')|
         (use cdr,nthcdr mode: exact direction: reverse)
         (use * mode: exact))
    ;NTHCDR(IDENT(M),N')=IDENTl(N',M-N')

6.  (ci id_main2)
    ;N'<M⊃NTHCDR(IDENT(M),N')=IDENT1(N',M-N')

7.  (ci id-mainl)
    ;(N<M⊃NTHCDR(IDENT(M),N)=IDENT1(N,M-N))⊃
    ;(N'<M⊃NTHCDR(IDENT(M),N')=IDENT1(N',M-N'))
```

```
a.  (ue (a |λn.n<m⊃nthcdr(ident(m),n)=ident1(n,m-n)|)
        proof-by-induction
        (part 1#1 (open minus ident)) * )
    ;∀N.N<M⊃NTHCDR(IDENT(M),N)=IDENT1(N,M-N)
    (label nthcdr_ident)
```

To finish the proof of the lemma we use again *Minusfact10...*

```
9.  (rw * (use minusfact10 mode: exact))
    ;∀N.N<M⊃NTHCDR(IDENT(M),N)=IDENT1(N,(M-N')')
```

. ..and then apply *Car Nthcdr*. In this last step we use the information about the length of the **ident** function (see Section 6.3.3) in simpinfo.

```
    ; labels : CAR,NTHCDR
    ;∀U N.N<LENGTH U⊃CAR NTHCDR(U,N)=NTH(U,N)

    ; labels : SIMPINFO
    ;∀N.LENGTH(IDENT(N))=N

10. (ue ((u. |ident m|)(n.n)) car-nthcdr (use * mode: always))
    ;N<M⊃N=NTH(IDENT(M),N)

11. (trw |∀n m.n<m⊃nth(ident m,n)=n| * )
    (label id-main) ■
```

Exercise. Prove Theorem 2 in the representation by functions

```
∀U.U⊗IDENT(LENGTH U)=U

∀U.INTO(U)⊃IDENT(LENGTH U)⊗U=U
```

using directly Theorem 2 (ii) (*Right Id*) and (iii) *(Left Id)*

```
∀U V W.ID(U)∧COMP(V,W,U)∧LENGTH W=LENGTH U⊃V=W

∀U V W.ID(U)∧PERM(W)∧LENGTH W=LENGTH U∧COMP(V,U,W)⊃W=V
```

6.5.3. Using Functions: Identity is a Permutation.

Using the above lemma, it easy to prove that the 'identity' list is a permutation, following the pattern of the proofs in Section 6.5.1.

Theorem 2 (i) (*Perm Ident*)

```
∀N.PERM(IDENT(N))
```

Proof.

```
     (proof perm_ident)

     ;only ontoness requires some help

1.   (assume |n<length ident(m)|)
     (label prm,idl)

2.   (rw * (open ident))
     ;N<M
     (label prm_id2)
```

Again notice that the fact *Length Ident*:

```
∀N.LENGTH (IDENT(N))=N
```

is in simpinfo.

```
3.   (derive |NTH(IDENT(M),N)=N| (* id-main))

4.  *(derive |member(nth(ident m,n),ident m)|
            (nthmember prm,idl) )

5.  (rw  *  (use -2 mode: exact))
     ;MEMBER(N,IDENT(M))

6.   (ci prm_id1)
     ;N<M⊃MEMBER(N,IDENT(M))

7.   (trw |∀n.perm(ident n)| (open perm into onto)
          (use id-main mode: always) * )
     ;∀N.PERM(IDENT(N))
     (label perm_ident) ■
```

6.5.4. **Using Functions: Right Identity.**

Using the lemma *Main Id* it is also easy to show that **ident** gives the right identity.

Theorem 2 (ii) (Right *Identity*)

```
∀U.U⊗IDENT(LENGTH U)=U
```

Remark. Example 9. We give two proofs of this Theorem, as evidence of our claim that a presentation through abstract lemmata (Proof 1) is more convenient than direct verification (Proof 2). The convenience is not simply in the fact that the first proof is shorter than the second: rather it lies in that we use the lemmata *Nth Compose* and *Main Id*, having many other applications, instead of proving a lemma, of interest only in this contest.

First Proof.

```
     (proof identity-right)

 1.  (rw perm,id (open perm onto))
     ;∀N.INTO(IDENT(N))∧(∀N1.N1<N⊃MEMBER(N1,IDENT(N)))

     ;labels:DEF,APPL,CONDITION
     ;∀U V.INTO(U)∧LENGTH U≤LENGTH V⊃DEF_APPL(V,U)

 2.  (ue ((u.|ident(length u)|)(v.u))
         def,appl,condition * (open lesseq))
     ;DEF_APPL(U,IDENT(LENGTH U))

     ;labels:NTH,COMPOSE
     ;∀V U N.DEF_APPL(V,U)∧N<LENGTH U⊃NTH(V⊗U,N)=NTH(V,NTH(U,N))

 3.  (ue ((u.|ident(length u)|)(v.u)(n.n)) nth_compose *
         (use id-main mode: exact))
     ;N<LENGTH U⊃NTH(U⊗IDENT(LENGTH U),N)=NTH(U,N)

     ;labels: EXTENSIONALITY
     ;∀U V.LENGTH U=LENGTH V∧(∀I.I<LENGTH U⊃APPL(U,I)=APPL(V,I))⊃U=V

 4.  (ue ((u.|u⊗ident(length u)|)(v.u)) extensionality (open appl)
         (use length-compose -2 *))
     ;U⊗IDENT(LENGTH U)=U
```

Notice that this proof is the same as that given in Section 6.5.1.

Second Proof. Without using the main lemma, we can prove Right Identity by proving first

```
     ∀n.n≤length u⊃u⊗ident1(length u-n,n)=nthcdr(u,length u-n)

     (proof identity-right)

 1.  (ue ((u.u)(n.|length ul)) trivial-nthcdr (open lesseq))
     ;NTHCDR(U,LENGTH U)=NIL

 2.  (trw |u⊗ident1(length u,0)=NTHCDR(u,length u)| (open identl compose)
          (use * mode: exact))
     ;U⊗IDENT1(LENGTH U,O)=NTHCDR(U,LENGTH U)
     (label irl)

 3.  (assume |n≤length(u)⊃u⊗ident1(length u-n,n)=nthcdr(u,length u-n)|)
     (label ir-hyp)

 4.  (assume |n'≤length u|)(label ir2)

 5.  (derive |u⊗ident1(length u-n,n)=nthcdr(u,length u-n)|
             (ir-hyp ir2 succ_lesseq_lesseq))
     (label ir3)

 6.  (derive |length u-(n')<length u| (minusfactll less-lesseqsucc ir2))
```

```
     (label ir4)

 7.  (derive |(length u-n')'=length u-n| (minusfact10 less,lesseqsucc ir2))
     (label ir5)

 8.  (trw |u⊗ident1(length u-(n'),n')=nthcdr(u,length u-(n'))|
          (open ident 1 compose)
          (use nthcdr,car,cdr ue: ((u.u)(n.|length u-(n')|)) mode: exact)
          ir4 ir5 ir3)
     ;U⊗IDENT1(LENGTH U-N',N')=NTHCDR(U,LENGTH U-N')
     ; deps : (IR,HYP IR2)

 9.  (ci ir2)
     ;N'≤LENGTH U⊃U⊗IDENT1(LENGTH U-N',N')=NTHCDR(U,LENGTH U-N')

 10. (ci ir,hyp)

 11. (ue (a |λn.n≤length(u)⊃u⊗ident1(length u-n,n)=nthcdr(u,length u-n)|)
         proof-by-induction
         (part 1#1 (open minus)) irl * )
     ;∀N.N≤LENGTH U⊃U⊗IDENT1(LENGTH U-N,N)=NTHCDR(U,LENGTH U-N)
```

The theorem follows immediately:

```
 12. (ue (n |length ul) * (open lesseq nthcdr) (use n-less-n))
     ;U⊗IDENT1(0,LENGTH U)=U

 13. (trw |u⊗ident(length u)=u| (open ident) * )
     ;U⊗IDENT(LENGTH U)=U
     (label identity-right) ■
```

□

6.5.5. Using Functions: Left Identity.

Similarly. by applying the Main Lemma for Identity, we can prove that ident gives the left identity by following the pattern of the proof in Section 6.5.1.

Theorem 2 (iii) (Left Identity)

```
     ∀U.INTO(U)⊃IDENT(LENGTH U)⊗U=U
```

Proof.

```
     (proof identity-left)

 1.  (assume lint0 u|)
     (label il_1)

 2.  (ue ((u.u)(v.|ident(length u)|))
          def_appl_condition
          * (open lesseq))
     ;DEF,APPL(IDENT(LENGTH U),U)
```

```
(label il_2)

3.  (rw il_1 (open into>>
    ;∀N.N<LENGTH U⊃NATNUM(NTH(U,N))∧NTH(U,N)<LENGTH U

4.  (ue ((v.|ident(length u)|)(u.u)) nth,compose il_2 *
        (use id-main ue: ((n.|nth(u,n)|)(m.|length u|)) ))
    ;∀N.N<LENGTH U⊃NTH(IDENT(LENGTH U)⊗U,N)=NTH(U,N)

5.  (ue ((u.|ident(length u)⊗u|)(v.u)) extensionality
        (sortcomp il_2 length-compose *) (open appl))
    ;IDENT(LENGTH U)⊗U=U

6.  (ci il_1)
    ;INTO(U)⊃IDENT(LENGTH U)⊗U=U
    (label identity-left) ■
```

It is completely clear that, by abstacting the main property of identity, we obtain a uniform treatment of all the parts of theorem 2 and greatly simplify the proofs. Actually the present version is even more elegant than that using predicates, since the espression **u⊗w** is easier to read than comp(v,u,w) (for humans as well as for computer programs).

6.6. Theorem 3: the Inverse of a Permutation.

6.6.1. Using Predicates: the Inverse of a Permutation is a Permutation.

Theorem3 (i) (*InvPerm*)

VU V.PERM(U)∧INV(V,U)∧LENGTH V=LENGTH U⊃PERM(V)

The proof of this theorem is obtained by expanding the definitions and making appropriate substitutions, in the style of Theorems 1 (i) and 2 (i). We give it in the Appendix.

6.6.2. Using Predicates: the Right Inverse Theorem.

Theorem 3 (ii) (*Right Inverse*)

VU V W.PERM(W)∧INV(U,W)∧COMP(V,W,U)∧LENGTH U=LENGTH W⊃ID(V)

Proof. Assume that w is a permutation (line 1), v is the result of composing w and u (line 4), where u is the inverse of w (line 2), and u is of the same length as w (line 3). We need to see that for all m < length(v), nth(v,m) = m (line 14).

The key point is the application of the lemma *NthFstposition* (line 13). To prepare it, we have only to expand the definitions and perform the right substitutions.

nth(v,m) = nth(w,nth(u,m)) (by line 7) if m<length(v)=length(u)

= nth(w,fstposition(w,m))) (line 10)

= m (line 13)

```
;the theorem right inverse
(proof  inverse-right)

1.  (assume |perm w|)
    (label invrl)

2.  (assume |inv(u,w)|)
    (label invr2)

3.  (assume |length u=length w|)
    (label invr3)

4.  (assume|comp(v,w,u)I)
    (label invr4)

5.  (rw invrl (open perm onto into))
    ;(∀N.N<LENGTH W⊃NATNUM(NTH(W,N))∧NTH(W,N)<LENGTH W)∧
    ;(∀N.N<LENGTH W⊃MEMBER(N,W))
    (label invr5)

6.  (rw invr2 (open inv))
    ;∀N.N<LENGTH U⊃NTH(U,N)=FSTPOSITION(W,N)
    (label invr6)

7.  (rw invr4 (open comp))
    ;LENGTH V=LENGTH U∧(∀N.N<LENGTH U⊃NTH(V,N)=NTH(W,NTH(U,N)))
    (label invr7)

8.  (assume |m<length v|)
    (label invr8)

9.  (rw * (use invr7 mode: exact))
    ;M<LENGTH U
    (label invr9)

10. (trw |nth(v,m)=nth(w,fstposition(w,m))| (invr7 *)
        (use invr6 mode: always direction: reverse))
    ;NTH(V,M)=NTH(W,FSTPOSITION(W,M))
    (label invr10)

11. (rw invr9 (use invr3 mode: exact))
    ;M<LENGTH W

12. (derive |member(m,w)| (invr5 *))

    ;labels: NTH_FSTPOSITION
```

```
     ;∀U N.MEMBER(N,U)⊃NTH(U,FSTPOSITION(U,N))=N

13. (rw invr10 (use nth_fstposition * mode: always))
     ;NTH(V,M)=M

14. (ci invr8)
     ;M<LENGTH V⊃NTH(V,M)=M

15. (trw | id(v) | (open id) * )
     ;ID(V)

16. (ci (invrl invr2 invr4 invr3))
     ;PERM(W)∧INV(U,W)∧COMP(V,W,U)∧LENGTH U=LENGTH W⊃ID(V)
     (label inv,right) ■
```

6.6.3. Using Predicates: the Theorem Left Inverse.

Theorem 3 (iii) (*Left Inv*)

$$\forall U\ V\ W.\mathrm{PERM}(W)\wedge \mathrm{INV}(U,W)\wedge \mathrm{COMP}(V,U,W)\wedge \mathrm{LENGTH}\ W=\mathrm{LENGTH}\ U\supset \mathrm{ID}(V)$$

Proof. Assume that w is a perm. that u is the inverse of w, v is the result. of composing u and w, and the length(w) = length(u). We need to prove that

$$\forall n.n<\mathrm{length}(v)\supset \mathrm{nth}(v,n)=n.$$

Assume that n < length(v). After expanding the definitions we know that.

$$\mathrm{length}(v) = \mathrm{length}(w),$$

so n< length(w) and n < length(u). Similarly, all members of w are natural numbers less than length(u) (lines 9. 13). So the sorts are verified and we can apply the definition of composition to get

$$\mathrm{nth}(v,n)=\mathrm{nth}(u,\mathrm{nth}(w,n))$$

(line 14), and the definition of inverse to obtain

$$\mathrm{nth}(u\otimes w,n)=\mathrm{fstposition}(w,\mathrm{nth}(w,n))$$

(line 15).

We want to conclude that

$$\mathrm{fstposition}(w,\mathrm{nth}(w,n))=n.$$

This need not be true if in w there are several occurrences of the n-th element. However. w is a permutation. By the pigeon hole principle w is inject ive; we can apply the lemma *FstpositionNth* (lines 8, 16) and obtain the desired conclusion (line 19).

```
    (proof compose-inverse-left)
1.  (assume |perm(w)|)
    (label invl,1)

2.  (assume |inv(u,w)|)
    (label invl,2)

3.  (assume |comp(v,u,w)|)
    (label invl,3)

4.  (assume |length(w)=length(u)|)
    (label invl,4)

5.  (rw invl,2 (open inv))
    ;∀N.N<LENGTH U⊃NTH(U,N)=FSTPOSITION(W,N)
    (label invl,5)

6.  (rw invl,1 (open perm onto into))
    ;(∀N.N<LENGTH W⊃NATNUM(NTH(W,N))∧NTH(W,N)<LENGTH W)∧
    ;(∀N.N<LENGTH W⊃MEMBER(N,W))
    (label invl,6)
    ;deps: (INVL,1)

7.  (rw invl,3 (open comp))
    ;LENGTH V=LENGTH W∧(∀N.N<LENGTH V⊃NTH(V,N)=NTH(U,NTH(W,N)))
    (label invl,7)

8.  (derive |∀n.n<length w⊃fstposition(w,nth(w,n))=n|
    (fstposition,nth perm,injectivity uniqueness,injectivity
     invl,1 invl,6))
    (label invl,8)
    ;deps: (INVL,1)

9.  (rw invl,6 (use invl,4 mode: exact))
    ;(∀N.N<LENGTH U⊃NATNUM(NTH(W,N))∧NTH(W,N)<LENGTH U)∧
    ;(∀N.N<LENGTH U⊃MEMBER(N,W))
    (label invl,9)
    ;deps: (INVL,1 INVL_4)

10. (assume |n<length v|)
    (label invl,10)

11. (rw * (use invl,7 mode: always))
    ;N<LENGTH W
    (label invl,11)
    ;deps: (INVL,3 INVL,10)

12. (rw * invl,4)
    ;N<LENGTH U
    (label invl_12)
    ;deps: (INVL,3 INVL,4 INVL,10)

13. (derive |natnum(nth(w,n))∧nth(w,n)<length u| (invl,9 *))
    (label invl,13)
```

```
;deps: (INVL,I INVL,3 INVL,4 INVL,IO)

14. (derive |NTH(V,N)=NTH(U,NTH(W,N))| (invl,7 invl,IO))
    (label invl_14)
    ;deps: (INVL_3 INVL_10)

15. (rw invl_14 (use invl,5 ue: ((n.|nth(w,n)|)) invl_13 mode: exact))
    ;NTH(V,N)=FSTPOSITION(W,NTH(W,N))
    (label invl,15)
    ;deps: (INVL,I INVL,2 INVL,3 INVL,4 INVL,IO)

    ;*want to apply the lemma fstposition,nth

16. (rw invl_15 (use invl,8 invl_11 mode: always))
    ;NTH(V,N)=N
    ;deps: (INVL,I INVL,2 INVL,3 INVL,4 INVL,IO)

    ;and so V is the identity function

17. (ci invl_10)
    ;N<LENGTH V⊃NTH(V,N)=N
    ;deps: (INVL-I INVL,2 INVL,3 INVL,4)

18. (trw |id v| (open id) * )
    ;ID(V)
    ;deps: (INVL-I INVL,2 INVL,3 INVL,4)

19. (ci (INVL,I INVL_2 INVL,3 INVL,4))
    ;PERM(W)∧INV(U,W)∧COMP(V,U,W)∧LENGTH W=LENGTH U⊃ID(V)
    (label inv_left) ∎
```

6.6.4. Using Functions: the Lemma Main Inv.

We follow the same strategy for the proof of the facts about the *inverse* operation. First we prove the main extensional property of *inverse* (compare with the definition of inv, Section 6.2.1):

Lemma 6.6. (*Main Inv*)

```
∀U N.PERM U∧N<LENGTH U⊃NTH(INVERSE U,N)=FSTPOSITION(U,N)
```

and then we follow the proof of Theorem 3 in Sections 6.6.1, 6.6.2 and 6.6.3 .

Proof of the Main Lemma. We show first that if u is a permutation. then

```
∀n.n<length u⊃nthcdr(inverse(u),n)=invers1(u,n,length u-n).
```

```
(proof inverse-main)

I.  (assume |perm u|)
    (label inv_main1)
```

We need to check that fstposition has the proper value on the intended domain.

```
2.  (rw inv_main1 (open perm onto))
    ;INTO(U)∧(∀N.N<LENGTH U⊃MEMBER(N,U))
    (label inv_main2)

3.  (ue ((u.u)(y.n)) posfacts)
    ;(NULL FSTPOSITION(U,N)⊃¬MEMBER(N,U))∧
    ;(MEMBER(N,U)⊃NATNUM(FSTPOSITION(U,N)))

4.  (derive |n<length u⊃¬null fstposition(u,n)| (inv_main2 *))
    (label inv_main3)
```

Next we give the inductive argument for our sublemma:

```
5.  (assume |n<length u⊃
            nthcdr(inverse(u),n)=invers1(u,n,length u-n) |)
    (label inv_main5)

6.  (assume |n'<length u|)
    (label inv_main6)

7.  (derive |n<length u| (* succ,less-less))
    (label inv_main7)

8.  (derive |¬null fstposition(u,n)| (inv_main3 inv_main7))
    (label inv_main9)
```

We use *Minusfact10* to expand the definition of invers1 in the right member of the equality.

```
9.  (rw inv_main5
        (use inv_main7 inv_main9)(open invers1)
        (use minusfact10 mode: always))
    (label inv_main10)
    ;NTHCDR(INVERSE(U),N)=FSTPOSITION(U,N).INVERS1(U,N',LENGTH U-N')
    ;deps: (INV_MAIN1 INV_MAIN5 INV_MAIN6)
```

We use *Cdr Nthcdr* to conclude the inductive step:

```
    ;labels: CDR_NTHCDR
    ;∀U N.CDR NTHCDR(U,N)=NTHCDR(U,N')

10. (ue ((u.|inverse u|)(n.n)) cdr_nthcdr (use * mode: exact))
    ;INVERS1(U,N',LENGTH U-N')=NTHCDR(INVERSE(U),N')
    ;deps: (INV_MAIN1 INV_MAIN5 INV_MAIN6)

11. (ci inv_main6)
    ;N'<LENGTH U⊃INVERS1(U,N',LENGTH U-N')=NTHCDR(INVERSE(U),N')

12. (ci inv_main5)

13. (ue (a |λn.n<length u⊃nthcdr(inverse(u),n)=invers1(u,n,length u-n)|)
        proof-by-induction  (part 1#1 (open inverse minus)) * )
    ;∀N.N<LENGTH U⊃NTHCDR(INVERSE(U),N)=INVERS1(U,N,LENGTH U-N)
    ;deps: (INV_MAIN1)
```

The main lemma follows. We use again *Minusfact10* to expand the definition of invers1...

```
14. (rw * (use minusfact10 mode: exact) (open inversl)
        (use inv_main3 mode: always))
    ;∀N.N<LENGTH U⊃
    ;NTHCDR(INVERSE(U),N)=FSTPOSITION(U,N).INVERS1(U,N',LENGTHU-N')
    ;deps: (INV_MAIN1)
```

...and then *Car Nthcdr*.

```
    ;labels: CAR-NTHCDR
    ;∀U N.N<LENGTH U⊃CAR NTHCDR(U,N)=NTH(U,N)

15. (ue ((u.|inverse(u)|)(n.n)) car-nthcdr
        (use * lengthinverse inv_main1 mode: always))
    ;N<LENGTH U⊃FSTPOSITION(U,N)=NTH(INVERSE(U),N)
    ;deps: (INV_MAIN1)

16. (ci inv_main1)
    ;PERM(U)⊃(N<LENGTH U⊃FSTPOSITION(U,N)=NTH(INVERSE(U),N))

17. (derive |∀u n.perm u∧n<length u⊃nth(inverse u,n)=fstposition(u,n) I * )
    (label inv,main) ∎
```

Exercise. Prove Theorem 3 in the representation by functions

```
∀U.PERM(U)⊃U⊗INVERSE(U)=IDENT(LENGTH(U))

∀U.PERM(U)⊃INVERSE U⊗U=IDENT(LENGTH U)
```

using directly Theorem 3 (ii) (*Right Inverse*) and (iii) (*Left Inv*)

```
VU V W.PERM(W)∧INV(U,W)∧COMP(V,W,U)∧LENGTH U=LENGTH W⊃ID(V)

VU V W.PERM(W)∧INV(U,W)∧COMP(V,U,W)∧LENGTH W=LENGTH U⊃ID(V)
```

6.6.5. Using Functions: the Inverse of a Permutation is a Permutation.

Theorem3(i) (*Perm Inverse*)

```
∀U.PERM(U)⊃PERM(INVERSE U))
```

Theorem3(ii) (*RightInverse*)

```
∀U.PERM(U)⊃U⊗INVERSE(U)=IDENT(LENGTH(U))
```

Theorem3(iii) (*LeftInverse*)

```
∀U.PERM(U)⊃INVERSE U⊗U=IDENT(LENGTH U)
```

Proof of Theorem 3 (i). The first part of the Theorem is the proof of the following fact, *Inv Into*:

```
∀U.PERM(U)⊃INTO(INVERSE(U))

(proof inverse-perm)

1. (assume |perm(u)|)
   (label inv,pl)

2. (rw * (open perm onto))
   ;INTO(U)∧(∀N.N<LENGTH U⊃MEMBER(N,U))
   (label inv_p2)

3. (ue ((u.u)(y.n)) posfacts)
   ;(NULLFSTPOSITION(U,N)⊃¬MEMBER(N,U))∧
   ;(MEMBER(N,U)⊃NATNUM(FSTPOSITION(U,N)))

4. (derive |∀n.n<length u⊃
           natnum fstposition(u,n)∧fstposition(u,n)<length u|
           (inv_p2 * pos_length))
   (label inv_p3)

5. (derive |∀n.n<length u⊃
           nth(inverse u,n)=fstposition(u,n)|
           (inv,main inv,pl))
   (label inv_p4)

6. (rw inv_p3 (use * mode: always direction: reverse))
   ;∀N.N<LENGTH U⊃NATNUM(NTH(INVERSE(U),N))∧NTH(INVERSE(U),N)<LENGTH U

7. (trw (into inverse(u)| *
        (open into) (use lengthinverse inv,pl mode: exact))
   ;INTO(INVERSE(U))
   (label into-inverse)

8. (ci inv,pl)
   ;PERM(U)⊃INTO(INVERSE(U))
   (label inv_into)
```

The second part of the theorem is the proof that `inverse(u)` is onto, still under the assumption that `perm(u)` (line 1).

```
9. (rw inv,pl (open perm into onto) )
   ;(∀N.N<LENGTH U⊃NATNUM(NTH(U,N))∧NTH(U,N)<LENGTH U)∧
   ;(∀N.N<LENGTH U⊃MEMBER(N,U))
   (label inv_p10)

10. (derive |length inverse(u)=length u| (inv,pl lengthinverse))
    (label inv_p11)

11. (assume |n<length inverse(u)|)
```

```
    (label inv_p12)

12. (rw * (use inv_p11 mode: exact))
    ;N<LENGTH U
    (label inv_p13)
```

We can apply the main property of the inverse function...

```
13. (ue (n |nth(u,n)|) inv_p4 (use inv_p10 * mode: always))
    ;NTH(INVERSE(U),NTH(U,N))=FSTPOSITION(U,NTH(U,N))
    (label inv_p14)
```

. . . the consequence of the Pigeon Hole principle. . .

```
14. (derive linj u| (inv_p1 perm_injectivity))
```

. ..the basic fact *Fstposition Nth...*

```
15. (derive |fstposition(u,nth(u,n))=n|
    (fstposition,nth uniqueness,injectivity * inv_p10 inv_p13))

16. (rw inv_p14 (use *))
    ;NTH(INVERSE(U),NTH(U,N))=N
    (label inv_p15)
```

. ..and the lemma *Nthmember...*

```
17. (derive |natnum nth(u,n)∧nth(u,n)<length inverse(u)|
            (inv_p10 inv,pll inv_p13))

18. (trw |member(nth(inverse u,nth(u,n)),inverse u)|
         (nthmember *))
    ;MEMBER(NTH(INVERSE(U),NTH(U,N)),INVERSE(U))
```

. ..to conclude:

```
19. (rw * (use inv_p15))
    ;MEMBER(N,INVERSE(U))
    ;deps: (INV_P1 INV_P12)

20. (ci inv_p12)
    ;N<LENGTH (INVERSE(U))⊃MEMBER(N,INVERSE(U))
    (label  onto-inverse)

21. (trw |perm(inverse u)| (open perm onto)
         into-inverse onto-inverse)
    ;PERM(INVERSE(U))

22. (ci inv_p1)
    ;PERM(U)⊃PERM(INVERSE(U))
    (label  perm_inverse) ∎
```

The proofs of the other parts of Theorem 3 are given in the Appendix.

7. Conclusion.

The remarks made in the Introduction and in the text, especially in Section 6.1, are relevant to the heuristics of automatic theorem proving and apply at three stages of the enterprise of mechanically representing mathematical facts.

■ . First, the choice of a representation determines the basic strategy of proof. It is certainly reasonable to search for a representation that allows simple recursive definitions of the basic objects and hence relatively simple proofs by induction on those recursive definitions. For this reason our representation using association lists is particularly attractive. However, there may be other reasons suggesting a different representation. In our case, we considered the representation by lists of numbers since it has the property of uniqueness.

Since EKL uses higher order language, it does not restrict us to a particular kind of representation: if recursive definitions are not available, or not convenient, we may give abstract definitions and carefully organize the argument so that appropriate mathematical or logical principles apply. As a very simple example, discussing the choice of predicates or of functions in the representation PERMP and PERMF we weighted two approaches: explicit definitions, derivations by logic inferences and term substitutions directed by the user *versus* recursive definitions, proofs by induction and logic inference replaced by rewriting. We considered advantages and limitations of the two methods and saw how a judicious combination of them may give the best results. At the end of Section 6.1, we outlined the optimal choice of definitions and the most effective proof strategy.

Moreover, EKL allows us to prove abstract mathematical facts that are independent of the particular representation: the Pigeon Hole principle was proved in second order arithmetic and then applied to different representations. In general there is no doubt that an essential advantage in proving correctness of programs is given by access to abstract mathematical knowledge.

■ Even when the main strategy of proof is chosen, different choices may be possible for the Lemmata. One can use EKL as an heuristic aid and try to find a proof by trial and error, reduce the task to some lemmata, try to prove the Lemmata, etc. (An example is given in the proof *Lengthinverse*, Section 6.3.4 .) A warning has to be made against this procedure: EKL will be extremely helpful in reminding us of many details we usually take for granted, but we are not yet ready to dismiss pencil and paper as obsolete: indeed it will save us a lot of time to work out a fairly detailed proof by 'pencil and paper' *before* starting our interaction with EKL.

Let us say that a proof is 'trivial' when the recursive definition of the basic objects and the statement of the theorem determine not only the main strategy of proof of the theorem, but also a natural choice of the lemmata and strategies for their proofs. Presumably, for such 'trivial' proofs some development of Boyer and Moore's techniques will allow entirely automatic heuristics.

There is no reason to think that given a simple recursive definition of some basic objects, one will always find 'trivial' proofs by induction. Often the choice of the Lemmata will not be obvious. Sometimes the lemmata suggested by the recursive definition of the objects and by the statement of the theorem by no means are the most convenient or the most perspicuous.

Consider for instance our definition of `inverse`, using `invers1` and `fstposition`: the functions involved here cannot be considered extremely difficult as LISP programs. However, there is room for discussion on how to choose and to prove the lemmata. Indeed, as we argued in the text, the best choice seems to be to consider the abstract properties of the functions `ident` and `inverse`, and prove them as lemmata. These properties are immediately recognized when we formulate the identity function and the operation of function inversion more abstractly as predicates. They are not the ones that come to mind first if we try to prove the theorems by expanding the definitions.

■ Finally, there is still room for choice at the stage of performing single inductive proofs, when the necessary lemmata have been proved. One may try to obtain the proof in a single line, by rewriting or expand the proof by using explicitly the logic decision procedure in the style of Natural Deduction. The heuristics of single proofs has been extensively discussed in the Conclusion of Part I, Section 2.13.

Remark. Example 10. To consider the different options available in carrying on a relatively long proof, let's look at the problem of formalizing the Lemma in Section 1.4 in full generality. We want to prove that for any $f : A \rightarrow B$ with A and B finite sets of the same cardinality, if f is a surjection then f is also an injection. We express this statement in our fragment of Set Theory, using our (higher order) formalization of arithmetic. The following is an outline of the project. The details are left to the reader as an exercise.

i) Formulate the Set Theoretic notions of map, surjection, injection and bijection, and use function abstraction and application to define function composition. For instance:

```
(decl (f g h) (type: |ground→ground|))

(decl map (type: |@f⊗@a⊗@a→truthval|))
(define map |∀g a b.map(g,a,b)≡(∀xv.xv∈a⊃g(xv)∈b)|)
(label mapdef)

(decl compmap (type: |@g⊗@g→@g|)(infixname: ⊗⊗)
      (bindingpower: 960))
(define compmap |∀f g.f⊗⊗g=(λxv.f(g(xv)))|)
(label compmapdef )
```

The fact that a set a has finite cardinality n can be expressed as

```
∀a n.fincard(a,n)≡∃f.bijection(f,segm(n),a)
```

(where segm(n) denotes $\mathbf{N}_n$).

The inverse image of an element y under a function g is

```
∀G A YV.INVIM(G,YV)= XXV.G(XV)=YV
```

ii) Apply the Pigeon Hole principle to maps $g : \mathbf{N}_n \rightarrow \mathbf{N}_n$. Formally, prove *Ontomap Injmap*

```
∀G N.ONTOMAP(G,SEGM(N),SEGM(N))⊃INJMAP(G,SEGM(N),SEGM(N))
```

A way to do this is to define a recursive functional card that counts the intersection of a set a with the set $\mathbf{N}_n$:

```
(define card |∀a n.card(a,0)=0∧
                  card(a,n')=if a(n) then card(a,n)' else card(a,n)|
     inductive-definition)
```

The next step is to prove the analogues for card of the properties of mult. The Pigeon Hole principle now takes the form

```
∀SETSEQ N.DISJOINT(SETSEQ,N)⊃
        ((∀M.M<N⊃1≤CARD(SETSEQ(M),N))⊃
         (∀M.M<N⊃1=CARD(SETSEQ(M),N)))
```

Let `setseq(m)` be `invim(g,m)`: by the properties of the inverse image and of surjective maps we obtain

```
∀G N.ONTOMAP(G,SEGM(N),SEGM(N))⊃
    (∀M.M<N⊃1=CARD(INVIM(G,M),N))
```

An argument by contradiction, counting cardinalities, gives *Ontomap Injmap*.

iii) Reduce the problem for arbitrary finite sets to the problem for sets of numbers. Namely, show that given an *onto* function $g : A \to B$ and suitable bijections $f_A : \mathbf{N}_n \to A$, $f_B : B \to \mathbf{N}_n$, there is a (finite) onto function $f : \mathbf{N}_n \to \mathbf{N}_n$ such that the diagram

$$\begin{tikzcd} A \arrow[r, "g"] & B \arrow[d, "f_B"] \\ \mathbf{N}_n \arrow[u, "f_A"] \arrow[r, "f"] & \mathbf{N}_n \end{tikzcd}$$

commutes. f is an onto function over $\mathbf{N}_n$, hence *Ontomap Injmap* applies. This involves some abstract properties of maps between sets. Conclude:

```
∀G A B N.FINCARD(A,N)∧FINCARD(B,N)∧ONTOMAP(G,A,B)⊃INJMAP(G,A,B)
```

From this general application of the Pigeon Hole principle one can derive the corresponding statements using various representations of finite functions. In the representation by association lists one can show

```
1.  ∀ALIST.PERMUTP(ALIST)⊃
      ONTOMAP(λX.APPALIST(X,ALIST),MKLSET(DOM(ALIST)),MKLSET(RANGE(ALIST)))

2.  ∀ALIST.PERMUTP(ALIST)⊃FINCARD(MKLSET(DOM(ALIST)),LENGTH (DOM(ALIST)))

3.  ∀ALIST.PERMUTP(ALIST)⊃
      BIJECTION(λX.APPALIST(X,ALIST),MKLSET(DOM(ALIST)),MKLSET(DOM(ALIST)))
```

and derive the familiar result *Permutp Injectp*

```
∀ALIST.PERMUTP(ALIST)⊃INJECTP(ALIST)
```

by using 1,2,3 and some fact about `appalist`. For instance, the following fact may be useful:

```
∀ALIST N.UNIQUENESS(DOM(ALIST))∧N<LENGTH (DOM(ALIST))⊃
          APPALIST(NTH(DOM(ALIST),N),ALIST)=NTH(RANGE(ALIST),N)
```

Clearly the above alternative route to prove *Permutp Injectp* is elegant and attractive, since the general facts may be applied in other contexts.

iv) In the same vein, one could want to do the entire project at a more abstract level. Namely, one can use EKL to check that bijections over *any* (not necessarily finite) set, with composition of functions, form a group. Facts about composition and the identity functions are easy, and one may use the Axiom of Choice to show that every bijection has a right inverse, and some **categorical** properties of mappings of sets to conclude that the right inverse is a two sided inverse.

The Axiom of Choice, i.e. the statement

```
(∀X.∃Y.A(X,Y))⊃(∃F.∀X.A(X,F(X)))
```

is built in EKL: whenever we have obtained the line

```
;(∀X.∃Y.A(X,Y))
```

we can ask EKL to define a suitable function `fv`

```
(define fv |∀x.A(x,f(x)) | * )
```

In the case of finite sets, by using *iii)* one can restrict oneself to prove that surjections form a group. Also the corresponding fact, say, for the representation of finite functions by association lists, can be obtained by proving the following facts:

(I) If f is a bijection over a finite set A, then there exists an association list $\texttt{alist}_f$ that represents f, i.e. such that for all elements x of A,

$$f(x) = \texttt{appalist(x,alist}_f\texttt{)}.$$

(II) If f_1 and f_2 are functions for which $f_2 \circ f_1$ is defined, and alist 1, alist2 represent f_1, f_2, respectively, then composition of functions is represented by 'composition' of association lists, i.e.

$$(f_2 \circ f_1)(x) = \texttt{appalist(alist1} \; \infty \; \texttt{alist2,x)}.$$

This approach is certainly very efficient and elegant. Here, however, we see that there may be a price to be paid for efficiency: by general considerations we only show the *existence* of an inverse function. We do not obtain the verification that a particular LISP program represents the inverse permutation. In logical terms, we verify that our LISP structures satisfy the axioms of groups using the axioms

$$\forall xyz.x \circ (y \circ z) = (x \circ y) \circ z,$$

$$\exists z.\forall x.\exists y.x \circ z = z \circ x = x \land x \circ y = y \circ x = z,$$

rather than the universal axioms

$$\forall xyz.x \circ (y \circ z) = (x \circ y) \circ z,$$

$$\forall x.x \circ e = e \circ x = x,$$

$$\forall x.x \circ x^{-1} = x^{-1} \circ x = e.$$

If the purpose of the project is mechanical verification of correctness of programs, the verification of a given program representing inversion of permutations has to be done separatly.

Despite our emphasis on the use of abstract mathematical tools, the approach to verification of properties of programs that has been followed in this paper could be described as the approach 'from below': given simple LISP programs performing some mathematical constructions, prove 'directly' the properties of programs that correspond to the mathematical facts in question.

The efficiency obtained by working with abstract notions suggests a different approach to mechanical program verification: working 'from above', we may formally prove facts with a maximum of generally and abstraction; only at the end we apply the result to the concrete programs. □

In conclusion, our experiment shows that, even when (i) our mathematical objects have simple recursive definitions, (ii) the proofs require no sophisticated methods and (iii) the heuristics itself appears mechanizable for some part of the proofs, it is still convenient to organize the subject through abstract lemmata, rather than to use direct proofs every time.

In Proof Theory procedures of Normalization play an essential role. Roughly speaking, when logical constants and mathematical entities can be appropriately defined according to the pattern Introduction – Elimination, it is possible to define a normal form for proofs and to find procedures that transform every proof into one in normal form.

In such procedures a sequence of inferences that first establish a general lemma and then apply it to particular conditions is considered a 'detour'. Such sequence must be simplified in favor of a longer but more direct proof. *From the point of view of Proof Theory it is essential to establish the possibility of normalization.* Important properties of mathematical systems can be established by these methods.

However, *Normalization does not seem to be the optimal strategy for proof checking.* In formalizing relatively large areas of knowledge it seems necessary to follow the opposite strategy, namely to search for suitable abstract Lemmata applicable to different situations.

Using Kreisel's words:

> 'The particular strategy of organizing an area of knowledge, which serves us here as a model, is the style of Bourbaki: one looks for a *few* definitions and key theorems that lead to easy solutions of *many* problems. (No one proof in Bourbaki is long).'
>
> *G.Kreisel* [1981]

APPENDIX

8. APPENDIX.

8.1. A Summary of Natural Deduction.

The aim of the following notes is to remind the reader of the basic features of Gentzen's system of Natural Deduction not only because of its historical importance in the design of EKL and of related systems (e.g. FOL), but also since some familiarity with Natural Deduction may be useful in constructing EKL proofs. The reader already familiar with the subject may want to skip this section.

A *Natural Deduction System* is a formal system that allows one to derive a formula as consequence of a list of formulas, the *assumptions*, and to eliminate formulas from a given assumption list. One of the deduction rule in a Natural Deduction system, given a derivation of B from a set of assumptions of the form A, allows one to construct a derivation of $A \supset B$, where (some of) the assumptions A have been *discharged* (see the $\supset$ - Introduction rule below).

By contrast a *Hilbert style axiomatic system* allows one only to derive logical consequences of certain formulas, regarded as axioms. We need a *metatheorem* to prove that in certain conditions if B is provable from a set of axioms $\mathcal{S}$ together with the axiom A then there is a proof of $A \supset B$ from $\mathcal{S}$ only (*Deduction Theorem).*

The most successful system of Natural Deduction was defined by Gentzen and later improved by Prawitz [1965, 1971]. In the Prawitz formulation, we are given a language with a symbol $\perp$ for falsity and the usual connectives and quantifiers $\wedge, \vee, \supset, \forall$ and $\exists$. Negation is defined: $\neg A$ is $A \supset \perp$. Moreover we use two disjoint sets of symbols for free and bound variables. A system of Natural Deduction is specified by *rules of inference* and *rules of deduction.*

A derivation is a finite tree of formulas (with 'leaves' at the top), where

(i) the top formulas ('leaves') are the assumptions,

(ii) the bottom formula is the conclusion,

(iii) every formula not at the top is derived by a rule of inference from the subderivation immediately above it and

(iv) a deduction rule associates to each occurrence of a formula a set of *open assumptions*, i.e. the set of assumptions, which the formula in question *depends on.*

Often in the literature the deduction rules are not explicitely specified, but the reader can easily fill in the details. (Actually, dealing with finite trees, an effective specification is always possible.) A useful convention is to divide assumptions of the same form into *assumption classes*, to mark with the same label an assumption class and the inference by which the assumption class is discharged.

A rule of inference has the form: *If* $\prod_1, \dots, \prod_n$ *are derivations of* $C_1, \dots C_n$, *respectively, then*

$$\begin{array}{ccc} \prod_1 & & \prod_n \\ C_1 & \dots & C_n \\ \hline \end{array}$$

is a derivation of C, under certain conditions on the form of the C_i's and Π_i's. The formulas C_i are called the *premises* and C the *conclusion* of the inference.

Thus the set of rules of inference, together with the clause

Every formula is a derivation of itself,

gives an inductive definition of derivations.

The reader will recognize the usual rules of introduction and elimination of the logical connectives and quantifiers in the figures below. More specifically, in each of the figures below, if the symbol(s) above the line denote derivation(s) of the indicated formula(s), then the displayed symbols denote a derivation of the formula below the line.

$\wedge$-Introduction

$$\frac{\begin{array}{c}\Pi_1\\ A\end{array}\qquad\begin{array}{c}\Pi_2\\ B\end{array}}{A\wedge B}$$

$\wedge$-Elimination

$$\frac{\begin{array}{c}\Pi\\ A\wedge B\end{array}}{A}\qquad\qquad\frac{\begin{array}{c}\Pi\\ A\wedge B\end{array}}{B}$$

$\supset$-Introduction

$$\frac{\begin{array}{c}[A]\\ \Pi\\ B\end{array}}{A\supset B}$$

$\supset$-Elimination

$$\frac{\begin{array}{c}\Pi_1\\ A\supset B\end{array}\qquad\begin{array}{c}\Pi_2\\ A\end{array}}{B}$$

$\vee$-Introduction

$$\frac{\begin{array}{c}\Pi\\ A\end{array}}{A\vee B}\qquad\qquad\frac{\begin{array}{c}\Pi\\ B\end{array}}{A\vee B}$$

$\vee$-Elimination

$$\frac{\begin{array}{c}\\ \Pi_1\\ A\vee B\end{array}\qquad\begin{array}{c}[A]\\ \Pi_2\\ C\end{array}\qquad\begin{array}{c}[B]\\ \Pi_3\\ C\end{array}}{C}$$

$\forall$-Introduction

$$\frac{\begin{matrix}\Pi_0\\ A(a)\end{matrix}}{\forall x.A(x)}$$

$\forall$-Elimination

$$\frac{\begin{matrix}\Pi\\ \forall x.A(x)\end{matrix}}{A(t)}$$

where no open assumption of Π_0 contains the free variable a. t is any individual term.

$\exists$-Introduction

$$\frac{\begin{matrix}\Pi\\ A(t)\end{matrix}}{\exists x.A(x)}$$

$\exists$-Elimination

$$\frac{\begin{matrix}\\ \Pi_1\\ \exists x.A(x)\end{matrix}\qquad\begin{matrix}[A(a)]\\ \Pi_2\\ C\end{matrix}}{C}$$

where the free variable a does not occur in C, in $\exists x.A(x)$, or in any open assumption of Π_2 different from $A(a)$. t is any individual term.

In any elimination rule the *first* premise, containing the symbol to be 'eliminated', is called the *major premise*. The other premise(s) (if any) are called *minor premise(s)*.

The following rules for negation are needed to formalize Intuitionistic Logic and Classic Logic.

$\bot_I$ (Intuitionistic)

$$\frac{\begin{matrix}\Pi\\ \bot\end{matrix}}{A}$$

$\bot_C$ (Classic)

$$\frac{\begin{matrix}[\neg A]\\ \Pi\\ \bot\end{matrix}}{A}$$

Under most rules of deduction the open assumptions associated with the conclusion of an inference are the union of the sets of open assumptions associated with the premises, with the following exceptions:

i) in $\supset$ - Introduction, the assumption class $[A]$ is discharged;

ii) in $\vee$ - Elimination, the assumption classes $[A]$ of Π_2 and $[B]$ of Π_3 are discharged;

iii) in $\exists$ - Elimination, the assumption class $[A(a)]$ of Π_2 is discharged;

iv) in the Classical Rule of Negation, the assumption class $[\neg A]$ is discharged.

Deduction rules can be specified by writing the set of open assumptions (followed by some symbol, e.g.'$\vdash$') before each formula occurrence in the derivation. Thus, using Greek letters for sets of assumptions, we can write the rules for disjunction as

$\vee$-Introduction

$$\dfrac{\begin{array}{c}\Pi\\ \Gamma\vdash A\end{array}}{\Gamma\vdash A\vee B} \qquad \dfrac{\begin{array}{c}\Pi\\ \Gamma\vdash B\end{array}}{\Gamma\vdash A\vee B}$$

$\vee$-Elimination

$$\dfrac{\begin{array}{c}\Pi_1\\ \Gamma\vdash A\vee B\end{array} \qquad \begin{array}{c}\Pi_2\\ \Delta_1\cup\{A\}\vdash C\end{array} \qquad \begin{array}{c}\Pi_3\\ \Delta_2\cup\{B\}\vdash C\end{array}}{\Gamma\cup\Delta_1\cup\Delta_2\vdash C}$$

Warning. A formulation of Natural Deduction along these lines would be only a typographical variant of the above system and *not* a form of Calculus of Sequents, a related but *conceptually different* logical calculus.

The restrictions on free variables for $\forall$-Introduction and $\exists$-Elimination establish a relation between free variables and rules. When performing transformations on proof it is convenient to have a different free variable for each application of such a rule. This can be handled by introducing a special list of variables, called *eigenvariables* or *parameters*, to be used in association with $\forall$-Introduction and $\exists$-Elimination.

One can prove that every derivation can be transformed into an equivalent one in which the eigenvariable associated with a $\forall$-I or $\exists$-E application occurs only in the ancestors of the conclusion of such rule. (*Lemma on parameters*, Prawtz [1965], p. 29).

A formal system that distinguishes between variables and parameters may be sometimes cumbersome, although the main idea is simple. In the top level language of EKL the distinction is not required (see rules about dependencies below).

The system **M**, containing only the rules for $\wedge$, $\vee$, $\supset$, $\forall$ and $\exists$, formalizes *Minimal Logic*. The system **I**, given by **M** plus the Intuitionistic Negation Rule $\bot_I$, is *(Heyting) Intuitionistic Logic*. The system **C**, given by **M** plus the Classical Negation Rule, is *full Classical Logic*. We write $\Gamma\vdash_I A$ ($\Gamma\vdash_C A$) to indicate that A is derivable from the formulas in Γ in the system for Intuitionistic (Classical) Logic.

Example 1.

$$\dfrac{\dfrac{\dfrac{\overset{(3)}{\neg(A\vee B)} \qquad \dfrac{\overset{(1)}{A}}{A\vee B}\,\vee\text{-I}}{\bot}\,\supset\text{-E}}{\neg A}\,\supset\text{-I, (1)} \qquad \dfrac{\dfrac{\overset{(3)}{\neg(A\vee B)} \qquad \dfrac{\overset{(2)}{B}}{A\vee B}\,\vee\text{-I}}{\bot}\,\supset\text{-E}}{\neg B}\,\supset\text{-I, (2)}}{\neg A\wedge\neg B}\,\wedge\text{-I}$$

(Here '$\supset$-I','$\supset$-E' stand for '$\supset$-Introduction','$\supset$-Elimination', etc. With the above specification of the deduction rules, we obtain a derivation $\neg(A \vee B) \vdash_I \neg A \wedge \neg B$.

Example 2.

$$\dfrac{\overset{(4)}{\neg A \vee B} \qquad \dfrac{\dfrac{\dfrac{\overset{(1)}{\neg A} \qquad \overset{(2)}{A}}{}\ \supset\text{-E}}{B}\ \bot_I}{A \supset B}\ \supset\text{-I},(2) \qquad \dfrac{\overset{(3)}{B}}{A \supset B}\ \supset\text{-I}}{A \supset B}\ \vee\text{-E},(1),(3)$$

This is a derivation $\neg A \vee B \vdash_I A \supset B$. Notice that we can infer $A \supset B$ from B *even* if the assumption class $[A]$ is empty—i.e. A is not an open assumption, which B depends on.

Example 3.a)

$$\dfrac{\overset{(2)}{\exists y \forall x.A(x,y)} \qquad \dfrac{\dfrac{\dfrac{\overset{(1)}{\forall x.A(x,b)}}{A(a,b)}\ \forall\text{-E}}{\exists y.A(a,y)}\ \exists\text{-I}}{\forall x \exists y.A(x,y)}\ \forall\text{-I}}{\forall x \exists y.A(x,y)}\ \exists\text{-E},(1)$$

Here we assume that $\exists y \forall x.A(x,y)$ does not contain a, b.

Example 3.b) What restrictions must C satisfy in order for the following to be a correct derivation?

$$\dfrac{\overset{(3)}{\exists y \forall x.A(x,y)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\overset{(2)}{C} \qquad \dfrac{\overset{(1)}{\forall x.A(x,b)}}{A(a,b)}\ \forall\text{-E}}{C \wedge A(a,b)}\ \wedge\text{-I}}{A(a,b)}\ \wedge\text{-E}}{\exists y.A(a,y)}\ \exists\text{-I}}{\forall x \exists y.A(x,y)}\ \forall\text{-I}}{\forall x \exists y.A(x,y)}\ \exists\text{-E},(1)$$

Example 4. *(i)* The rule $\perp_C$ is derivable in the system **I** plus the axiom $A \vee \neg A$.
(ii) $\vdash_C A \vee \neg A$. It is easy to see that in the proof we must assume $\neg(A \vee \neg A)$.

It will not be difficult to convince the reader that the derivation in Example 3(a) is 'better' than that in Example 3(b). Not only Example 3(a) is shorter, but also the two successive steps of Introduction and Elimination of $\wedge$ in Example 3(b) do not give us any interesting information: the formula C is simply irrelevant for the derivability of the conclusion from the premises.

This simple remark can be extended to other rules and gives the main idea of Normalization Theorem, one of the most interesting result of Proof Theory.

The occurrence of a formula in a derivation is called *maximal* if it is at the same time the conclusion of an Introduction and the major premise of an Elimination rule (necessarily, of the same logical symbol). A maximal formula can be removed by a *reduction*, a transformation of the given derivation that consists essentially of removing the two steps, Introduction and Elimination. Here we give the list of reductions.

$\wedge$-Reduction

$$\dfrac{\dfrac{\begin{matrix}\Pi_1 & & \Pi_2 \\ A_1 & & A_2\end{matrix}}{A_1 \wedge A_2}\,\wedge\text{-I}}{[A_i]}\,\wedge\text{-E}$$
$$\Pi_3$$

where $i = 1$ or 2, is reduced to

$$\begin{matrix}\Pi_i \\ [A_i] \\ \Pi_3\end{matrix}$$

⊃-Reduction

$$
\begin{array}{c}
\dfrac{\dfrac{\begin{array}{c}[A]\\ \Pi_1\\ B\end{array}}{A \supset B}\,\supset\text{-I} \qquad\qquad \begin{array}{c}\Pi_2\\ A\end{array}}{[B]}\,\supset\text{-E}\\
\Pi_3
\end{array}
$$

is reduced to

$$
\begin{array}{c}
\Pi_2\\
[A]\\
\Pi_1\\
[B]\\
\Pi_3
\end{array}
$$

∨-Reduction

$$
\begin{array}{c}
\dfrac{\dfrac{\begin{array}{c}\Pi_0\\ A_i\end{array}}{A_1 \vee A_2}\,\vee_i\text{-I} \qquad \begin{array}{c}(1)\\ [A_1]\\ \Pi_1\\ C\end{array} \qquad \begin{array}{c}(2)\\ [A_2]\\ \Pi_2\\ C\end{array}}{[C]}\,\vee\text{-E},\ (1),(2)\\
\Pi_3
\end{array}
$$

where $i = 1$ or 2, is reduced to

$$\prod_0$$

$$[A_i]$$

$$\prod_i$$

$$[C]$$

$$\prod_3$$

The following derivation assume that $\prod_1 (a)$ satisfies the restriction for the $\forall$-Introduction and contains a only at some ancestors of $A(a)$. (One can show that any derivation can be transformed into an equivalent derivation satisfying the last condition.) Then

$\forall$-Reduction

$$\begin{array}{l} \prod_1(a) \\ \\ \dfrac{\dfrac{A(a)}{\forall x.A(x)}\,\forall\text{-I}}{[A(t)]}\,\forall\text{-E} \\ \\ \prod_2 \end{array}$$

is reduced to

$$\prod_1(t)$$

$$[A(t)]$$

$$\prod_2$$

where $\prod_1(t)$ is the result of replacing everywhere t for a in $\prod_1(a)$.

The following derivations assume that $\prod_2(a)$ satisfies the restriction for the $\exists$-Elimination and contains a only at some descendants of $A(a)$.

$\exists$-**Reduction**

$$\begin{matrix} \dfrac{\dfrac{\begin{matrix}\prod_1\\ A(t)\end{matrix}}{\exists x.A(x)}\,\exists\text{-I} \qquad\qquad \begin{matrix}(1)\\ [A(a)]\\ \prod_2(a)\\ C\end{matrix}}{[C]}\,\exists\text{-E}, (1) \\ \prod_3 \end{matrix}$$

is reduced to

$$\begin{matrix} \prod_1 \\ [A(t)] \\ \prod_2(t) \\ [C] \\ \prod_3 \end{matrix}$$

where $\prod_2(t)$ is the result of replacing everywhere t for a in $\prod_2(a)$.

Example 5. The following derivation formalizes a common procedure: first prove a general lemma $(\forall x.(A(x) \supset B(x)))$ and then apply it to particular cases.

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{matrix}(1)\\ [A(a)]\\ \prod_1(a)\\ B(a)\end{matrix}}{A(a) \supset B(a)}\,\supset\text{-I}, (1)}{\forall x.(A(x) \supset B(x))}\,\forall\text{-I}}{A(t) \supset B(t)}\,\forall\text{-E} \qquad\qquad \begin{matrix}\prod_2\\ A(t)\end{matrix}}{B(t)}\,\supset\text{-E}$$

A derivation is said to be *normal* if it does not contain maximal formulas.

Normalization Theorem. Every derivation in the system for Int uitionistic Logic can be transformed into one in normal form.

The reader may have noticed that the normal derivations in Examples 1, 2 and 3(a) are, in some sense, 'among the best possible', but with a difference: the derivation in Example 1 is, in some sense, 'unique', whereas the others are not. Indeed in Example 2 we could permute the the $\supset$-I applications and the $\vee$-E and still obtain a normal derivation. Similarly in Example 3(a) we could permute the $\vee$ - I and the $\exists$-E. Of course these remarks could be made precise, but our examples suggest that uniqueness of the normal form may fail in a nontrivial sense.

Normal derivations have a very interesting structure. Their analysis requires some technical notions. Let $\prod$ be a derivation. A *path* in $\prod$ is a sequence $A_1, \ldots, A_n$ of formula occurrences in $\prod$ such that:

1) A_1 is a top formula that is not discharged by $\vee$-Elimination or $\exists$-Elimination;

2) A_n is either the endformula of $\prod$ or the minor premise of an $\supset$-Elimination; A_i, for $i < n$, is not the minor premise of $\supset$-Elimination;

3) for $i < n$, one of the following cases applies:

(i) A_i is a premise of an Introduction, of a Negation rule, of a $\wedge$-Elimination, $\forall$-elimination, or the major premise of $\supset$-Elimination, and A_{i+1} is the conclusion of that inference;

(ii) A_i is a minor premise of an $\vee$-Elimination or of an $\exists$-Elimination and A_{i+1} is the conclusion of that inference;

(iii) A_i is the major premises of an $\vee$-Elimination of of an $\exists$-Elimination and A_{i+1} is an assumption discharged by that inference.

Example 6. In the derivation of Example 1

— $\langle A, A \vee B \rangle$ is a path, (i.e. the formula occurrence labeled (1) and the one immediately below it);

— $\langle \neg(A \vee B), \perp, \neg A, \neg A \wedge \neg B \rangle$ is a path (starting with the leftmost occurrence of $\neg(A \vee B)$ labeled (3)).

In the derivation of Example 2

— the occurrence of A labeled (2) is a path;

— $\langle \neg(A \vee B), 7.4, \perp, B, A \supset B, A \supset B \rangle$ is a path (starting with the formula occurrence labeled (4) and continuing with the one labeled (1)).

In a path there may be consecutive occurrences of the same formula, e.g. the minor premise and the conclusion of an $\exists$-Elimination. A *segment* σ in a pat h of $\prod$ is a sequence $\langle A_1, \ldots, A_k \rangle$ of occurrences of (the same) formula, such that either $1 = k$ or, if $k > 1$, the following conditions are satisfied:

i) for $i = 2, \ldots, k$, A_i is the consequence of an $\vee$-Elimination or of $\exists$-Elimination, and A_1 is not such a consequence;

ii) for $i = 1, \ldots, k-1$, A_i is the minor premise of an $\vee$-Elimination or of an $\exists$-Elimination and A_k is not such a premise.

Example 7. Every formula occurrence in Example 1 is a segment. In Example 2, the sequence $\langle A \supset B, A \supset C\rangle$ consisting of a minor premise and the conclusion of the final $\vee$-Elimination is a segment; the sequence $\langle A \supset B\rangle$ containing the endformula of Example 2 is not a segment.

With this terminology, one can prove the following theorem, giving a complete characterization of normal deductions in Intuitionistic Logic (see Prawitz [1965], pag. 53):

Theorem. *Let $\prod$ be a normal deduction in the Natural Deduction system for Intuitionistic Logic, let π be a path in $\prod$, and let $\sigma_1, \ldots, \sigma_n$ be the sequence of segments in π. Then there is a segment σ_i, called the* minimum segment *in π, which separates two (possibly empty) parts of π, called the* E-part *and* I-part *of π, with the properties:*

1) For each σ_j in the E-part (i.e. $j < i$), the last formula occurrence in σ_j is a major premise of an E-rule and the formula in σ_{j+1} is a subformula of the formula in σ_j.

2) If $i \neq n$, the formula in σ_i is a premise of an I-rule or of the $\perp_I$ Negation rule.

3) For each σ_k in the I-part (i.e. $i < k < n$), the last formula occurrence in σ_k is a premise of an I-rule and the formula in σ_k is a subformula of the formula in σ_{k+1}.

As a corollary of this analysis, one proves

Subformula Property. Every formula occurring in a normal derivation of A from Γ in the system of Natural Deduction for Intuitionistic Logic is a subformula either of A or of a formula in Γ.

The above result is fundamental in Proof Theory. For our purpose it is essential to notice that *if $\Gamma \vdash_I A$, then in the search for a normal derivation we need to consider only subformulas of A and of the formulas in Γ; moreover the above Theorem gives directions to build such a proof.* A normal deduction is in practice the best choice for a short proof. We may find it convenient to break a long derivation into lemmata either for the sake of readability or in order to highlight some important step in the argument.

By contrast, Example 4 (ii) shows that the Subformula Property fails for full Classical Logic. Indeed we do not have a Normalization for full Classical Logic. In order to overcome the difficulty, Gentzen introduced the Calculus of Sequents and Prawitz [1965,1971] proves Normalization for the formulation of Classical Logic without $\vee$ and $\exists$. However we will use the full system for Classical Logic, and not the Calculus of Sequents, so these results, despite their theoretical relevance, are beyond our immediate concern. For practical purpose, the reader may have noticed that *when an argument by contradiction is needed, there may be different ways to obtain one. A good choice of the formula to be contradicted is an essential step to obtain a readable* proof *and is not given by a mechanic procedure.*

Remark. Proofs by induction do not fit well in the pattern Introduction-Elimination of Natural Deduction: one may define what it means for the conclusion of an Induction Rule to be maximal and prove a Normalization Theorem for for first order the Natural Deduction system of Peano Arithmetic. (see Troelstra [1973]). The significance of the result, however, is reduced by the fact that in such system the Subformula Property does not hold. For higher order logic Prawitz proved Normalization Theorem (Prawitz [1968]). Again, the Subformula Property does not hold. In practice, if we apply some axiom of induction (or of a corresponding rule) in the context of higher order logic and recursive functionals of higher types, the simple form of normal deductions given by the Normalization Theorem for first order logic is necessarily lost.

As we shall see, we would like to let the computer do the logical steps of our proofs. To a certain extent, we succeeded in replacing logical steps by rewriting. However, a certain familiarity with Natural Deduction is important for the user of EKL. It suggests a safe procedure, though a lengthy one, to expand proofs. It may help us to understand what additional information is needed for the rewriting process to succeed.

8.2. Organization of the Files.

In practice, proofs are printed in electronically created files, that can be reached either directly by the user or automatically by EKL through the command

```
(get-proofs proofname).
```

Our proofs are distributed in the files described below.

NORMAL contains rewriters to normalize formulas.

NATNUM gives basic facts of arithmetic, i.e. addition, multiplication and ordering.

MINUS introduces more elementary arithmetic, including subtraction.

LISPAX contains the axioms of LISP.

ALLP allows to use the recursive predicate **allp** to replace bound quantifiers.

SET contains some notions of set theory.

LENGTH contains the definition of **length** and facts about it.

NTH contains the definitions of **nth**, **nthcdr**, **fstposition** and facts about them.

APPL contains the main definitions of application and permutation, in the two different representations, (I) using association lists, and (II) using lists of numbers

'SUMS contains the notions of finite union, finite sums and bound quantifiers **allnum** and **somenum**.

MULT contains the definition of of the function multiplicity.

PIGEON presents the proof of the pigeon-hole principle.

ALPIG contains the application of the pigeon hole to functions represented by association lists.

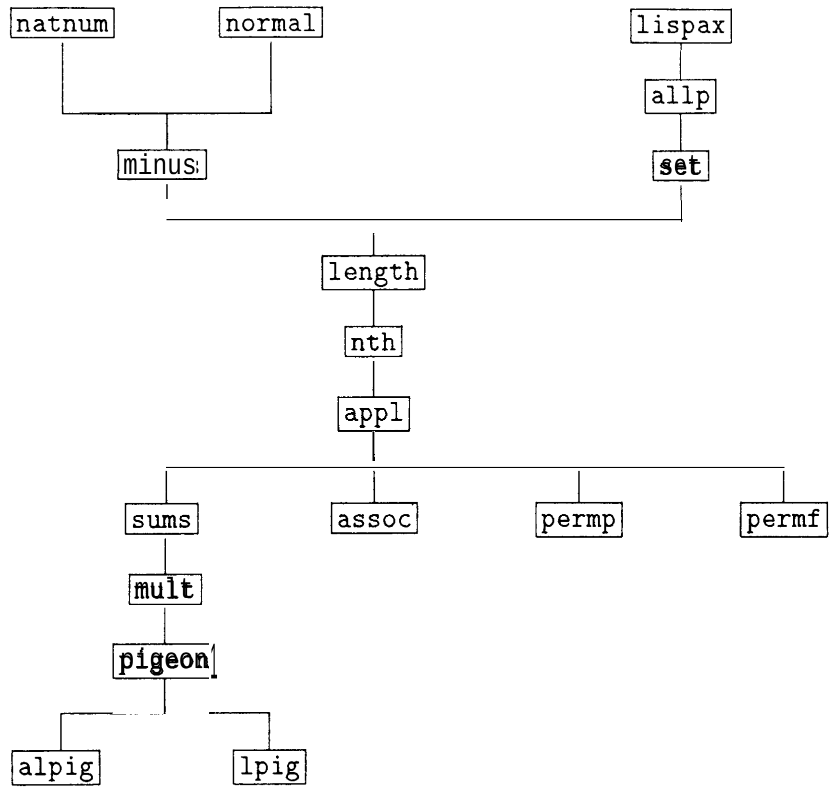
ALPIG contains an application of the pigeon hole to functions represented by lists of numbers.

ASSOC contains the definition of the operations of composition, identity and inversion of functions, represented as association lists (representation (I)) and proofs of all the facts about permutations.

PERMP contains the definitions of the operation of composition, identity and inversion of functions, using *predicates* in representation (II) and all the facts about permutations.

PERMF contains the definitions of composition, identity and inversion of functions, using *functions* in representation (II) and the corresponding proofs.

The dependency of the files is as follows



The reason for this organization of the files is to save memory when running the proofs. For the same reason we state our theorems as axioms, we "save" them for "quick-reference" to EKL and then we prove them.

One should not consider these details as merely 'administrative matter'. Quick access to stored information is very important in practice and the amount of memory involved even in easy proofs is considerable. Moreover, just as humans do not look at all the details when reading a proof (but are supposedly able to reconstruct them, if asked), so a computer program checking a proof should remember the relevant facts, not necessarily their proofs.

In the text most of the results are given with their proofs. Some facts of preliminary character are only quoted; their proofs can be found in this Appendix.

8.3. file NORMAL.

```
;propositional schemata, used by the rewriter to normalize expressions
(wipe-out)
(proof normal)
```

1.
```
(axiom |∀p q r.((p∨q)∧r)≡((p∧r)∨(q∧r))|)
(label normal)
```

2.
```
(axiom |∀p q r.(r∧(p∨q))≡((r∧p)∨(r∧q))|)
(label normal)
```

```
3.  (axiom |∀p q r.(r∧(p∨q))≡((p∧r)∨(q∧r))|)
    (label normal)

4.  (axiom |∀p q r.(p∨q⊃r)≡(p⊃r)∧(q⊃r)|)
    (label normal)

5.  (axiom  |∀p q.(¬(p∨q))≡((¬p)∧(¬q))|)
    (label demorgan)

6.  (axiom |∀p q.¬(p∧q)≡¬p∨¬q|)
    (label demorgan1)

    ;It would cause combinatorial explosion,to add these to simpinf 0, or to put everything,
    ;say,in conjunctive normal form. So we call them as rewriters when needed.

    ;a fea tricks

7.  (axiom |∀p q.p≡(q⊃p)∧(¬q⊃p)|)
    (label excluded-middle)

8.  (derive |∀p q r.(q⊃r)∧(if p then q else r)⊃r|)
    (label trans_cond)

    (save-proofs normal)
```

8.4. file NATNUM.

We collect here the most elementary facts of arithmetic, omitting their proofs. Our purpose is to have a collection of facts useful in various contexts, rather than to give a systematic treatment of elementary arithmetic from Peano Axioms. Our basic inductive principles include Second Order Induction Axiom and definition of primitive recursive functions and functionals of higher type.

```
    ;basic facts about arithmetic and proofs by Bellin

    (proof natnum)

1.  (decl lessp (type: |ground⊗ground→truthval|) (syntype: constant)
          (infixname: <) (bindingpower: 920))
2.  (decl add1 (type: |ground→ground|) (syntype: constant) (postfixname: I'|)
          (bindingpower: 975))
3.  (decl plus (infixname: |+|) (type: |ground⊗ground⊗ground*→ground|)
          (syntype : constant) (associativity: both) (bindingpower: 930))
4.  (decl times (type: |ground⊗ground⊗ground*→ground|) (syntype: constant)
          (infixname: |*|) (associativity: both) (bindingpower: 935))

5.  (decl (i j k n m) (sort: natnum) (type: ground))
6.  (decl (a b c set) (type: |ground→truthvalI))

    ;needed axioms on order

7.  (axiom |∀n.¬n<n|)
    (label irreflexivity-of-order)

8.  (axiom |∀n m k.n<m∧m<k⊃n<k|)
    (label trans ıtivity_of-order)

9.  (axiom |∀n.¬n<0|)
    (label zeroleast1)

    ;successor and order
```

```
10. (axiom |∀n.natnum(n')|)
    (label simpinfo)

11. (axiom |∀n.n<n'|)
    (label successor1) (label succfacts)

12. (axiom |∀n m.¬n<m⊃m<n'|)
    (label successor2) (label succfacts)

13. (axiom |∀n m.n'<m'≡n<m|)
    (label successorless)(label succfacts)

14. (axiom |∀n m.(n'=m')≡(n=m)|)
    (label successoreq) (label succfacts)

15. (axiom |∀n.¬n=0⊃0<n|)
    (label zeroleast2) (label succfacts)

16. (axiom |∀n.0<n'|)
    (label zeroleast3)(label succfacts)

17. (axiom |∀n.¬(n'=0)|)
    (label zero_not_successor)(label succfacts)

    ;definition of predecessor

18. (decl pred (type: |ground→ground|) (syntype: constant))
19. (defax pred |∀n.pred(n')=n|)
    (label pred_def)(label simpinfo)

20. (axiom |∀n.natnum pred n|)
    (label simpinfo)

    ;addition

21. (defax plus |∀n k.0+n=n∧k'+n=(k+n)'|)
    (label plusdef)(label simpinfo)(label plusfacts)

22. (axiom |∀n m.natnum(n+m)|)
    (label simpinfo)

23. (axiom |∀n.n+0=n|)
    (label simpinfo) (label plusfacts)

24. (axiom |∀n.1+n=n'∧n+1=n'|)
    (label simpinfo) (label plusfacts) (label plusdef1)

25. (axiom |∀n k.n+k'=(n+k)'|)
    (label simpinfo) (label plusfacts)

26. (axiom |∀n k m.(k+m=k+n)≡(m=n)|)
    (label lpluscan) (label plusfacts)

27  (axiom |∀n k m.(m+k=n+k)≡(m=n)|)
    (label rpluscan) (label plusfacts)

28. (axiom |∀n k.n+k=0≡n=0∧k=0|)
    (label addtozero) (label plusfacts)

    ;the effect of the following axiom is to force sums in basically normal
    ;form : the "simpler" terms will come first

29. (axiom |∀k n. k+n=n+k |)
    (label commutadd) (label plusfacts)

    ;multiplication
```

```
30. (defax times |∀n k.0*n=0∧n'*k=(n*k)+k|)
    (label timesdef)(label simpinfo) (label timesfacts)

31. (axiom |∀n m.natnum(m*n)|)
    (label simpinfo)

32. (axiom |∀n.n*0=0∧1*n=n∧n*1=n|)
    (label simpinfo) (label timesfacts)

33. (axiom |∀n k.n*k'=n*k+n|)
    (label timsucc) (label timesfacts)

34. (axiom |∀n k m.¬k=0⊃((k*m=k*n)≡(m=n))|)
    (label ltimescan) (label timesfacts)

35. (axiom |∀n k m.¬k=0⊃((m*k=n*k)≡(m=n))|)
    (label rtimescan) (label timesfacts)

36. (axiom |∀n m.n*m=m*n|)
    (label commutmult) (label timesfacts)

37. (axiom |∀n k.¬n=0⊃n*k=0≡k=0 )
    (label ltimestozero) (label timesfacts)

38. (axiom |∀n k.¬n=0⊃k*n=0≡k=0|)
    (label rtimestozero) (label timesfacts)

    ;distributivity

39. (axiom |∀n k m.n*(k+m)=n*k+n*m|)
    (label ldistrib) (label timesfacts) (label plusfacts)

40. (axiom |∀n m k.(m+k)*n=m*n+k*n|)
    (label rdistrib) (label timesfacts) (label plusfacts>

    ;inductive principles

    (proof induction)

1.  (axiom |∀a.a(0)∧(∀n.a(n)⊃a(n'))⊃(∀n.a(n))|)
    (label proof-by-induction)

2.  (decl npars (type: |ground*|))
3.  (decl ndf (type: |ground•ground*→ground*|))
4.  (decl zcase (type: |ground*→ground|))
    (axiom
      |∀ndf zcase ndef.
          (∃fun.(∀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

5.  (axiom |∀a2.(∀n m.a2(0,n)∧a2(n,0)∧(a2(n,m)⊃a2(n',n')))⊃∀n m.a2(n,m)|)
    (label proof-by-doubleinduction)

    ;general definitional principle for inductive functions

6.  (decl (arb arbl arb2) (type: |?arbitrary|))
7.  (decl indfn (type: |ground•@arb→@arb|))
8.  (decl (def_fun) (type: |ground→@arb|))

    ;this is the primitive recursive schema for definition on ALL
    ;higher type functionals:
    ;note the use of the variable type in declarations;
```

```
;in this way we can specialize to ANY type.
```

9.
```
(axiom
  |∀indfn arb.∃def_fun.∀n.def_fun(0)=arb∧
                          def_fun(n')=indfn(n,def_fun(n))|)
(label high-order-natnum-definition)

;well-foundedness
```

10.
```
(axiom |¬∃desc.∀n.desc(n')<desc(n)|)
(label infinite-descent)

 (save-proofs natnum)
```

8.4.1. More Arithmetic.

```
;proofs of facts of arithmetic
(wipe-out)
(get-proofs normal)
(get-proofs natnum)
(label simpinfo zero-not-successor) ;add these to simpinfo for now
(label simpinfo zeroleast1)
(label simpinfo successorless)
(label simpinfo successoreq)
(label simpinfo zeroleast3)

(proof lesseq)

;an easy consequence of the axioms in natnum
```

1.
```
(ue (a |λn.¬n=n'|) proof_by_induction)
(label simpinfo)(label successorfacts) ■
```

2.
```
(decl lesseq (type: |ground⊗ground→truthval|)(infixname: |≤|)
(bindingpoaer: 920))
```

3.
```
(define lesseq |∀m n.(m ≤ n)=(m=n∨m<n)|)
(label lesseqdef)

;successorlesseq
```

4,
```
(trw  ∀n m.n'≤m'≡n≤m| (open lesseq) )
(labe  successorlesseq) (label successorfacts) (label simpinfo) ■

;trans_lesseq
```

5.
```
(trw |∀n m k.n≤m∧m≤k⊃n≤k| (open lesseq) (use normal mode: always)
     transitivity-of-order)
;∀N M K.N≤M∧M≤K⊃N≤K
(label trans_lesseq) ■

;less_lesseq_fact1
```

6.
```
(trw |∀n m k.n<m∧m≤k⊃n<k| (open lesseq) (use normal mode: always)
     transitivity-of-order)
;∀N M K.N<M∧M≤K⊃N<K
(label less_lesseq_fact1) ■

;zeroleast
```

7.
```
(ue (a |λn.0≤n|) proof-by-induction (part 1 (open lesseq)))
;∀N.0≤N
```

```
    (label zeroleast) ■

    ;oneleastsucc

8.  (trw |0'≤n'| zeroleast)
    ;0'≤N'

9.  (rw * (nuse successorlesseq))
    ;1≤N'
    (label oneleastsucc) ■

    ;zero non less successor

10. (trw |m=0∧n'<m|)
    ;¬(M=0∧N'<M)

11. (derive |∀n m.n'<m⊃¬m=0| * )
    (label simpinfo)(label zero-non-less-successor)    ■

    ;a couple of very trivial facts
    ;succ_less_less

12. (trw |∀m n.m'<n⊃m<n| transitivity-of-order successor11
    (label succ-less-less) ■

    ;succ_lesseq_lesseq

13. (derive |M'=N⊃M<N| successor1)

14. (trw |∀m n.m'≤n⊃m≤n|(open lesseq)
        succ-less-less * (use normal mode : always))
    ;∀M N.M'≤N⊃M≤N
    (label succ_lesseq_lesseq) ■

    ;lesseq lesseq succ

15. (trw |∀n m.n≤m⊃n≤m'| (open lesseq) (use normal mode: always)
        (successor1 transitivity-of-order))
    (label lesseq_lesseq_succ) ■

    ; "m less succ of n" implies "m lesseq n"

16. (ue (a |λn.n<0'≡n≤0|) proof-by-induction
        (part 1 (open lesseq)))
    ;∀N.N<1≡N≤0

17. (ue (a2 |λn m.m<n'≡m≤n|) proof-by-doubleinduction * zeroleast)
    ;∀N M.M<N'≡M≤N
    (label less_succ_lesseq) ■

    ;"n less than m" implies "succ of n lesseq m"

18. (ue (a |λn.0<n≡0'≤n|) proof-by-induction
        zeroleast (part 1#1 (open lesseq)))
    ;∀N.0<N≡1≤N

19. (ue (a2 |λn m.n<m≡n'≤m|) proof-by-doubleinduction
        * (part 1#1#2 (open lesseq)))
    ;∀N M.N<M≡N'≤M
    (label less-lesseqsucc) ■

    ; "n lesseq m" and "m lesseq n" implies "n equal m"

20. (ue (a2 |λn m.n≤m∧m≤n⊃n=m|) proof-by-doubleinduction
        (part 1 (open lesseq) (use normal mode: always)) )
    ;∀N M.N≤M∧M≤N⊃N=M
```

```
(label leq_leq_eq) ■

;trichotomy
21. (rw zeroleast (open lesseq))

22. (ue (a2 |λn m.m<n∨m=n∨n<m|) proof-by-doubleinduction
        (use normal mode: always) * )
    ;∀N M.M<N∨M=N∨N<M
    (label trichotomy) ■
```

8.4.2. Subtraction.

```
    ;minus
    (proof minus)

1.  (decl minus (type: |ground⊗ground→ground|)(infixname: |-|)
    (bindingpower: 940))

2.  (define minus |∀m n.m-0=m∧m-(n')=pred(m-n)| inductive_definition)
    (label minusdef)

    ;minus sort

    ;the following proof works because pred is a total function

3.  (ue (a |λn.∀k.natnum(k-n)|) proof-by-induction
        (part 1 (open minus>>>
    ;∀N K.NATNUM(K-N)
    (label simpinfo) (label minus-sort)     ■

    ;minusfact3

4.  (ue (a |λn.n<m'⊃pred(m'-n)=m-n|) proof-by-induction
        (part 1 (open minus pred)) succ-less-less)
    ;∀N.N<M'⊃PRED(M'-N)=M-N

5.  (ue (a2 |λn m.n<m⊃0<m-n|) proof-by-doubleinduction (open minus)
        (use * mode: always) succ-less-less)
    ;∀N M.N<M⊃0<M-N
    (label minusfact ■

    ;minusfact5

6.  (ue (a |λn.0<n⊃pred(n)'=n|) proof-by-induction)
    ;∀N.0<N⊃PRED(N)'=N
    (label minusfact5)    ■

    ;successor minus

7.  (ue (a |λn.n<m'⊃m'-n=(m-n)' |) proof-by-induction
        (use -2 -3 successor1 succ-less-less mode: exact)
        (use * ue: ((n.|m-n|)) ) (part 1 (open minus pred)))
    ;∀N.N<M'⊃M'-N=(M-N)'

8.  (deriv e  |∀n m.n≤m⊃m'-n=(m-n)'|(* less_succ_lesseq))
    (label successor-minus)     ■

    ;pred_cancellation

9.  (trw |∀n m.n≤m⊃pred(m'-n)=m-n| successor-minus)
    ;∀N M.N≤M⊃PRED(M'-N)=M-N
    (label pred_cancellation) (label minusfact7)     ■
```

```
    ;minusfact10
10. (trw |∀n m.n<m⊃(m-n')'=m-n| (use minusfact ue: ((n.|m-n|)) )
        minusfact (open minus))
    ;∀N M.N<M⊃(M-N')'=M-N
    (label minusfact10)     ■

    ;minusfactll

11. (rw successor-minus (open lesseq) (use normal mode: always))

12. (derive |∀n m.n<m⊃m'-n=(m-n)'| * )

13. (ue (a |λn.0<n⊃pred(n)<n|) proof_by_induction successor11
    ;∀N.0<N⊃PRED(N)<N

14. (ue (a2 |λn m.m<n⊃n-(m')<n|) proof,by-doubleinduction
        (part 1 (use * -2 minusdef mode: always))
        (transitivity-of-order successor1))
    ;∀N M.M<N⊃N-M'<N
    (label minusfact11)     ■

    ;n less n

15. (rw successor-minus (open lesseq)(use normal mode: always))

16. (derive |∀n m.n=m⊃m'-n=(m-n)'| * )

17. (ue (a |λn.n-n=0|) proof-by-induction
        (part 1 (use * mode: always)(open minus)>>
    ;∀N.N-N=0
    (label simpinfo) (label n-less-n)     ■

    ;minusl

18. (ue ((n.n)(m.|n|)) successor-minus (open lesseq))

19. (ue (a |λn.0<n⊃n-(pred n)=1|) proof-by-induction (open pred) *
        (use successor-minus n-less-n mode: exact))
    (label minus1)     ■

    ;total subtraction

20. (ue (a |λn.m<n⊃m-n=0|) proof-by-induction (open minus lesseq)
        (use less-succ-lesseq mode: exact) (use normal mode: always))
    ;∀N.M<N⊃M-N=0

21. (trw |∀n m.m≤n⊃m-n=0| (open lesseq)
        (use normal mode: always) * n-less-n)
    ;∀N M.M≤N⊃M-N=0
    (label total-subtraction)     ■

    ;inequality law

22. (derive |∀k n.k<n'⊃n'-k=(n-k)'| (successor-minus less-succ-lesseq))

23. (ue (a2 |λn m.n<m⊃0<m-n|) proof-by-doubleinduction (open minus)
        (use * mode: always)(use succ_less_less))
    ;∀N M.N<M⊃0<M-N

24. (ue (a2 |λn m.k<n∧m<n-k≡m+k<n|) proof-by-doubleinduction
        (use * -2 mode: always))
    ;(∀N M.K<N∧M<N-K≡M+K<N⊃K<N'∧M<N-K≡M+K<N)⊃(∀N M.K<N∧M<N-K≡M+K<N)

25. (rw * (use less-succ-lesseq mode: exact)
        (open lesseq)(use normal mode: always))
```

```
;∀N M.K<N∧M<N-K≡M+K<N
(labelinequality-law)
```
■

The following two facts are needed in the induction step of the proof of the pigeon-hole principle.

Lemma. (*Add Lesseq*)

```
∀N K M.N≤M∧1≤K⊃N'≤M+K
```

The lower bound of a sum is the sum of the lower bounds. We use by double induction.

Proof.

```
;add_lesseq

1   (trw |∀n.0'≤n'| (use zeroleast))

2.  (rw * (nuse successorlesseq))
    ;∀N.1≤N'
```

The following line gives one base case, by a subordinate induction; the preceding line, with an automatic substitution of $n + k$ for n, proves the induction step for it.

```
3.  (ue (a |λn.0≤n∧1≤k⊃1≤n+k|) proof-by-induction * )
    ;∀N.0≤N∧1≤K⊃1≤N+K
```

The other base case reduces to a tautology, using the next line.

```
4.  (trw |∀n.n≤0⊃n=0| (open lesseq))
    ;∀N.N≤0⊃N=0
```

The induction step follows automatically from the line *Successorlesseq* (proof *min us*) which is in "simpinfo".

```
5.  (ue (a2 |λn m.n≤m∧1≤k⊃n'≤m+k|) proof-by-doubleinduction
        -2 (use * mode: always))
    ;∀N M.N≤M∧1≤K⊃N'≤M+K
    (label add_lesseq) ■
```

Lemma. (*Add One*)

```
∀K N M.1≤K∧N'=M+K∧N≤M⊃1=K∧N=M
```

If the sum of two variables equals the sum of the lower bounds, then the values of the variables must be their lower bounds.

Proof. Again we use double induction. One base case (i.e. $n = 0$) is also proved by double induction.

```
;add_one

1.  (ue (a2 |λm k.0'≤k∧0'=m+k⊃1=k∧0=m|) proof-by-doubleinduction
        (part 1 (open lesseq)(use normal mode: always>)>
    ;∀N M.1≤M∧1=N+M⊃1=M∧0=N
```

Here the other base case ($k = 0$) and the induction step are trivial, since the antecedent becomes false. In the other base case, when $m = 0$, we first rewrite $1 \leq k$ as $1 < k \lor 1 = k$; i.e. we "open" the symbol *lesseq* and then normalize. We obtain either a contradiction in the antecedent ($1 < k \wedge 1 = k$) or the desired result.

The next line is now easy: the base case $n = 0$ is the last line. In the other base case $m = 0$, we get $n = 0$ after opening $\leq$ (since $n < 0$ is impossible) and therefore also $1 = 0 + k$. The induction step follows from the lines *Successorlesseq*, (proof *MINUS*) and *Successoreq* and *Plusfacts* (proof *NATNUM*) that are in simpinfo.

```
2.  (ue (a 2 |λn m.1≤k∧n'=m+k∧n≤m⊃1=k∧n=m|) proof-by-doubleinduction
        (part 1#1#2 (open lesseq))
        (part 1#1#1 (use *)))
    ;∀N M.1≤K∧N'=M+K∧N≤M⊃1=K∧N=M
    (label add-one) ■
```

8.5. file LISPAX.

We define the basic functions of LISP and give their properties as axioms. We have basic principles of induction on lists and S-expressions and primitive recursive definition of LISP functions and higher order functionals.

```
    (proof lispax)

    ;;;declarations: note that t and nil are not declared - EKL knows about them
    ;;;since they are attached, we don't need to say things like null nil etc.

1.  (decl car (unaryname: car) (type: |ground→ground|) (syntype: constant)
    (bindingpower: 950))

2.  (decl cdr (unaryname: cdr) (type: |ground→ground| ) (syntype: constant)
    (bindingpower: 950))

3.  (decl atom (unaryname: atom) (type: |ground→truthval|) (syntype: constant)
    (bindingpower: 750))

4.  (decl null (unaryname : null) (type: |ground→truthval|) (syntype: constant)
    (bindingpower: 750))

5.  (decl listp (unaryname: listp) (type: |ground→truthval|) (syntype: constant)
    (bindingpower: 750))

6.  (decl alistp (unaryname: alistp) (type: |ground→truthval|)
    (syntype: constant)(bindingpower: 750))

7.  (decl sexp (unaryname: sexp) (type: |ground→truthval|) (syntype: constant)
    (bindingpower: 750))

8   (decl (u v a> (type: |ground|) (sort: |listp|))

9   (decl (x y z) (type: |ground|) (sort: |sexp|))

10. (decl (xa ya za) (type: |ground|) (sort: |atom|))

11. (decl (phi) (type: |ground→truthval|))

12. (decl cons (type: |(ground⊗ground)→ground|) (syntype: constant)
    (infixname: |.|)(prefixname: cons> (bindingpower: 850))

    ;;;basic axioms and sort info

13. (axiom |∀xa.sexp(xa)|)
    (label simpinfo)
```

```
14. (axiom |∀u.sexp u|)
    (label simpinfo)

15. (axiom |∀x u.listp x.u|)
    (label simpinfo)

16. (axiom |∀u.¬null u ⊃ listp cdr u|)
    (label simpinfo)

17. (axiom |∀u.¬null u ⊃ sexp car u|)
    (label simpinfo)

18. (axiom |∀x.¬atom x 3 sexp car xl>
    (label simpinfo)

19. (axiom |∀x.¬atom x ⊃ sexp cdr xl)
    (label simpinfo)

20. (axiom |∀x y.sexp x.y|)
    (label simpinfo)

21. (axiom |∀x y.¬atom x.y|)
    (label simpinfo)

22. (axiom |∀x u.¬null x.u|)
    (label simpinfo)

23. (axiom |∀u.null u ⊃ u = nil|)
    (label simpinfo)

24. (axiom |∀x y.car (x.y) = x|)
    (label simpinfo)

25. (axiom |∀x y.cdr (x.y) = y|)
    (label simpinfo)

26. (axiom |car nil =nil|)
    (label simpinfo)

27. (axiom lcdr nil = nil|)
    (label simpinfo)

28  (axiom |∀u.¬null u ⊃ (car u.cdr u=u)|)
    (label simpinfo) (label cons-car-cdr)

29. (axiom |∀x.¬atom x ⊃ (car x.cdr x=x) I>
    (label simpinfo) (label cons-car-cdr)

    ;;;induction

30. (axiom |∀phi.phi(nil)∧(∀x u.phi(u)⊃phi(x.u))⊃(∀u.phi(u))|)
    (label listinduction)

31. (decl pars (type: |ground*|))
32. (decl (df dfl df2) (type: |ground⊗ground*→ground*|))
33. (decl nilcase (type: |ground*→ground*|))
34. (axiom
      |∀df nilcase def.
          (∃fun.(∀pars x u.fun(nil,pars)=nilcase(pars)∧
                            fun(x.u,pars)=def(x,u,fun(u,df(x,pars)),pars)))|)
    (label listinductiondef)

35. (axiom
      |∀phi.(∀x.atom x ⊃ phi(x))∧(∀x y.phi(x)∧phi(y)⊃phi(x.y))⊃(∀x.phi(x))|)
    (label sexpinduction)
```

```
36. (axiom
      |∀atomcase defsexp df1 df2.∃fun.
          ∀pars x y z.
                (atom z ⊃
                  fun(z,pars)=atomcase(z,pars))∧
                (fun(x.y,pars)=
                  defsexp(x,y,fun(x,df1(x,y,pars)),fun(y,df2(x,y,pars)),pars))|)
    (label sexpinductiondef )

    ;a high order definition schema when above is insufficient

37. (decl (arb arb1 arb2) (type: |?arbitrary|))
38. (decl bigfun (type : |ground•ground•@arb•@arb→@arb|))
39. (decl (defined-fun atom-fun) (type: |ground→@arb|))

    ;this is the primitive recursive schema for definition on ALL
    ;higher type functionals:
    ;note the use of the variable type in declarations;
   ;in this way we can specialize to ANY type.

40. (axiom
      |∀bigfun atom_fun.∃defined_fun.
         ∀x y.(atom x ⊃
                defined_fun(x)=atom_fun(x))∧
              (defined_fun(x.y)=
                bigfun(x,y,defined_fun(x),defined_fun(y)))|)
    (label high-order-definition)

    ;;; lists of variable numbers of arguments don't require special treatment,
    ;;; since we have list types now

41. (decl list (type: |ground* → ground|) (syntype: constant))
42. (decl lst (type: |ground*|))

43. (axiom |list() = nil|)
    (label simpinfo)

44. (axiom |∀lst.listp(list(lst))|)
    (label simpinfo)

45. (axiom |∀x lst.list(x,lst) = x.list(lst)|)
    (label listdef)(label simpinfo)

    ;;; this is lisp's append. while it can be proved associative, it
    ;;; is convenient in proofs of other theorems to have it declared
    ;;; associative.

46. (decl append (type: |ground•ground•(ground*)→ground|) (syntype: constant)
      (associativity: both) (infixname: *) (bindingpower: 840))

47. (defax append |∀x u v.nil*v=v∧(x.u)*v=x.(u*v)|)
    (label appendef) (label simpinfo)

48. (axiom |∀u v.listp(u*v)|)
    (label simpinfo) (label listappend)

49. (axiom |∀u.u*nil=u|)
    (label simpinfo)

50. (axiom |∀x v.(x.nil)*v=x.v|)
    (label simpinfo)

    ;;;map functions on lists

51. (decl (allp somep) (syntype: constant) (type: |(@phi)•ground→truthval|))
```

```
52. (defax allp
       |∀phi x u.allp(phi,nil)∧
                allp(phi,x.u)=if phi(x) then allp(phi,u) else false|)
    (label allpdef)

53. (defax somep
       |∀phi x u.¬somep(phi,nil)∧
                somep(phi,x.u)=if phi(x) then true else somep(phi,u) |)
    (label somepdef)

54. (defax mapcar
       |∀fn x u.mapcar(fn,nil)=nil∧mapcar(fn,x.u)=fn(x).mapcar(fn,u)|)
    (label mapcardef)

55. (decl (alist) (type: ground) (sort: alistp))
56. (axiom |∀alist. listp alist|)
    (label simpinfo)

57. (axiom |∀u.alistp u ≡ (¬null u 3
                         ¬atom car uhatom car (car u)∧alistp(cdr u)) I )
    (label alistdef 1)

58. (axiom |∀xa y alist.alistp nil A alistp (xa.y).alist|)
    (label alistdef ) (label simpinf 0)

59. (decl assoc (type:  |ground•ground → ground|) (syntype: constant))

60. (defax assoc
       |∀x xa y alist.assoc(x,nil)=nil∧
                  assoc(x,(xa.y).alist)=(if x=xa
                                         then xa.y
                                         else assoc(x,alist)) I>
    (label assocdef )

61. (axiom |∀x alist.sexp assoc(x,alist)I )
    (label simpinf 0 )

62. (decl member (type: |ground•ground → truthval|) (syntype: constant))

63. (defax member |∀x y u. ¬member(x,nil)∧member(x,y.u)=(x=y∨member(x,u))|)
    (label memberdef )

64  (decl uniqueness (type: |ground→truthval|))

65. (defax uniqueness |∀u x.uniqueness nil A
                      (uniqueness(x.u)≡¬member(x,u)∧uniqueness(u))|)
    (label uniquenessdef)

66. (ue (phi |λu.sexp car(u)|) listinduction)
    (label simpinfo)

67. (ue (phi |λu.listp cdr(u)|) listinduction)
     label simpinfo)

    (save-proofs lispax)
```

8.5.1. file ALLP.

```
;properties of allp
(wipe-out)
(get-proofs lispax)
```

```
;proofs allp

(proof allpprop)

1.  (trw |∀phi x u.allp(phi,x.u)⊃phi(x)∧allp(phi,u)| (open allp))
    ;∀PHI X U.ALLP(PHI,X.U)⊃PHI(X)∧ALLP(PHI,U)
    (label allpfact) ■

2.  (ue (phi |λu.(∀y.member(y,u)⊃phi1(y))⊃allp(phi1,u)|)
        listinduction (open allp member) (use normal mode: always))
    ;∀U.(∀Y.MEMBER(Y,U)⊃PHI1(Y))⊃ALLP(PHI1,U)
    (label allp_introduction) ■

3.  (ue (phi |λu.member(x,u)∧allp(phi1,u)⊃phi1(x)|) listinduction
        (part 1 (open member allp) (use normal mode: always)))
    ;∀U.MEMBER(X,U)∧ALLP(PHI1,U)⊃PHI1(X)
    (label allp_elimination) ■

4.  (ue (phi |λu.∀a a1.allp(a,u)∧(∀x.a(x)⊃a1(x))⊃allp(a1,u)|) listinduction
        (open allp))
    ;∀U A A1.ALLP(A,U)∧(∀X.A(X)⊃A1(X))⊃ALLP(A1,U)
    (label allp_implication) ■

(proof someprop)

1.  (ue (phi |λu.member(y,u)∧phi1(y)⊃somep(phi1,u)|)
        listinduction (open somep member) (use normal mode: always))
    ;∀U.MEMBER(Y,U)∧PHI1(Y)⊃SOMEP(PHI1,U)

2.  (derive |∀u.(∃y.member(y,u)∧phi1(y))⊃somep(phi1,u)| *)

3.  (ue (phi |λu.somep(phi1,u)⊃(∃x.member(x,u)∧phi1(x))|)
        listinduction
        (part 1 (open member somep) (use normal mode: always) (der)))
    ;∀U.SOMEP(PHI1,U)⊃(∃X.MEMBER(X,U)∧PHI1(X))

4.  (derive |∀u.somep(phi1,u)≡(∃x.member(x,u)∧phi1(x))| (* -2))
    (label somepfact) ■
```

8.6. file SET.

```
;useful set theory
(wipe-out)
(get-proofs allp)

(proof sets)
;all urelements will be S-expressions
;all S-expressions will be urelements

1.  (decl (xv yv zv) (type: |ground|) (sort: urelement))
2.  (decl (av bv) (type: |ground→truthval|))

3.  (axiom |∀x.urelement x|)
    (label simpinfo)

4.  (axiom |∀xv.sexp(xv)|)
    (label simpinfo)

5.  (decl epsilon (type: |ground⊗@av→truthval|) (infixname: ∈) (bindingpower: 925))
6.  (define epsilon |∀av xv.xv∈av≡av(xv)|)
    (label epsilondef)
```

```
7.  (axiom |∀av bv.(∀xv.xv∈av≡xv∈bv)⊃av=bv|)
    (label set-extensionality)

8.  (decl intersect ion (type : |@av•@av→@av|)
          (infixname: ∩) (bindingpower: 950) (prefixname: intersection))
9.  (define intersection |∀av bv.av∩bv=λxv.(av(xv)∧bv(xv))|)
    (label interdef )

10. (decl union (type: |@av•@av→@av|)
          (infixname: ∪) (bindingpower: 950) (prefixname: union))
11. (define union |∀av bv.av∪bv=λxv.(av(xv)∨bv(xv))|)
    (label uniondef)

12. (decl inclusion (type: |@av•@av→truthval|)
          (infixname: ⊂) (bindingpower: 920) (prefixname: inclusion))
13. (define inclusion |∀av bv.av⊂bv≡∀xv.av(xv)⊃bv(xv)|)
    (label inclusiondef)

14. (defax emptyset |emptyset=λxv.false|)
15. (defax emptyp |∀av.emptyp(av)=∀xv.¬av(xv)|)

    ;the set of occurrences of an S-exp

16. (decl mkset (type: |ground→@av|))
17. (define mkset |∀x.mkset(x)=(λyv.yv=x)|)
    (label mkset_def)

    ;the set of members of a list

. 18. (decl mklset (type: |ground→@av|))
19. (define mklset |∀u.mklset(u)=λx.member(x,u)|)
    (label mklsetdef)

    (proof setfacts)

    ;fact about mkset and mklset

1.  (trw |∀u.member(y,u)⊃mkset(y)⊂mklset(u)|
         (open mkset mklset inclusion) (der))
    ;∀U.MEMBER(Y,U)⊃MKSET(Y)⊂MKLSET(U)
    (label mkset_mklset) ■

    ;double inclusion

2.  (ue ((av.av)(bv.bv)) set-extensionality (open epsilon))
    ;(∀XV.AV(XV)≡BV(XV))⊃AV=BV

3.  (derive |av⊂bv∧bv⊂av⊃av=bv| (*) (open inclusion))
    (label double-inclusion) ■
    (save-proofs set)
```

8.7. file LENGTH.

```
    ;facts about lengths of lists
    (get-proofs set)
    (get-proofs minus)

    (proof length)

    (decl length (type: Iground→ground|) (unaryname: length) )
2.  (define length |∀u x. (length nil=0)∧length(x.u)=(length u)' I
```

```
        (use listinductiondef))
    (label simpinfo) (label lengthdef)

3.  (ue (phi |λu.natnum length u|) listinduction (open length))
    ;∀U.NATNUM(LENGTH U)
    (label simpinfo) ■

4.  (ue (phi |λu.(length u=0)≡null u|) listinduction
        (open length> (use zero-not-successor>>
    ;∀U.LENGTH U=0≡NULL U
    (label simpinfo) ■

5.  (ue (phi |λu.length(u*v)=length u+length v|) listinduction
        (open append length))
    ;∀U.LENGTH (U*V)=LENGTH U+LENGTH v
    (label lengthadd) (label simpinfo)  ■

6.  (trw |length(x.nil)| (open length))
    ;LENGTH (X.NIL)=1
    (label simpinfo)  ■

7.  (derive |length(u)≤n∨n<length(u)| trichotomy (open lesseq))
    (label trichotomy2) ■

8.  (ue (phi |λu.member(y,u)⊃0<length u|) listinduction (open member))
    ;∀U.MEMBER(Y,U)⊃0<LENGTH U
    (label simpinfo)(label have-member)  ■

9.  (ue (phi |λu.member(y,u)⊃¬null u|) listinduction (open member))
    ;∀U.MEMBER(Y,U)⊃¬NULL U
    (label simpinfo)(label have_member1)  ■

    (save-proofs length)
```

8.8. file NTH: Some Appropriate Inductive Principles.

```
(wipe-out)
(get-proofs length)

;now we need to tie up natural numbers and s-expressions

(axiom |∀n.sexp n|)
(label simpinfo)

(axiom |∀n.¬null(n)|)
(label simpinfo)

(proof sets)
;all numbers will be urelements

(axiom |∀n.urelement n|)
(label simpinfo)

;forms of doubleinduction

(proof listinduction)

;a useful principle which follows from listinduction
;corresponds to a proof by cases arguments

(trw |∀phi.(phi(nil)∧∀x u.phi(x.u))⊃∀u.phi(u)| listinduction)
;∀PHI.PHI(NIL)∧(∀X U.PHI(X.U))⊃(∀U.PHI(U))
```

```
(label listcases)

;the next principle gives a convenient form for double induction on lists

(assume |∀u v x y.phi2(nil,u)∧phi2(u,nil)∧(phi2(u,v)⊃phi2(x.u,y.v))|)
(label dindass)

(ue (phi |λu .∀v .phi2(u,v)|) listinduction
    (use dindass) (use listcases ue: ((phi.|λv.phi2(x.u,v)|)) ))
;∀U V.PHI2(U,V)
;deps: (DINDASS)

(ci (dindass) * )
;(∀U V X Y.PHI2(NIL,U)∧PHI2(U,NIL)∧(PHI2(U,V)⊃PHI2(X.U,Y.V)))⊃(∀U V.PHI2(U,V))
(label doubleinduction) ■

;the next principle gives a form of double induction for lists and numbers

(assume |∀u n x.phi3(nil,n)∧phi3(u,0)∧(phi3(u,n)⊃phi3(x.u,n'))|)
(label dindassl)

(ue (phi |λu.∀n.phi3(u,n)|) listinduction
    (use dindassl) (use proof-by-induction ue: ((a.|λn.phi3(x.u,n)|)) ))
;∀U N.PHI3(U,N)
;deps: (DINDASS1)

(ci (dindassl) * )
;(∀U N X.PHI3(NIL,N)∧PHI3(U,0)∧(PHI3(U,N)⊃PHI3(X.U,N')))⊃(∀U N.PHI3(U,N))
(label doubleinductionl) ■
```

8.9. Nth.

```
;basic facts about nth
(proof nth)

1.  (decl nth (syntype: constant) (type: |ground⊗ground→ground|))

2.  (defax nth |∀x u n.nth(nil,n)=nil∧nth(u,0)=car u∧
                       nth(x.u,n')=nth(u,n)|)
    (label simpinfo) (label nthdef)

    ;prove by double induction the well-definedness of nth
    ;for the obvious range

3.  (ue (phi3 |λu n.sexp nth(u,n)|) doubleinductionl (open nth))
    ;∀U N.SEXP NTH(U,N)
    (label simpinfo)(label sexp_nth)   ■

    ;prove by double induction the membership of nth in the original list

4.  (ue (phi |λu.0<length u⊃member(nth(u,0),u)|) listinduction
        (open length nth member))
    ;∀U.0<LENGTH U⊃MEMBER(NTH(U,0),U)

    (ue (phi3 |λu n.n<length u J member(nth(u,n),u)|) doubleinductionl
        (open length nth) (use memberdef mode: always)(use *))
    ;∀U N.N<LENGTH U⊃MEMBER(NTH(U,N),U)
    (label nthmember)   ■
```

8.9.1. Member Nth.

```
;member_nth
(proof member-nth)

1.  (assume |(member(y,u)⊃(∃n.n<length u∧nth(u,n)=y))|)
    (label m_n1)

2.  (assume |y=x|)
    (label m_n2)

3.  (trw |0<length(x.u)∧nth(x.u,0)=y| (open nth) * )
    ;0<LENGTH (X.U)∧NTH(X.U,0)=Y

4.  (derive |∃n.n<length(x.u)∧nth(x.u,n)=y| * )
    (label m_n3)

5.  (assume |member(y,u)|)

6.  (define nv |nv<length u∧nth(u,nv)=y| (m_n1 *))

7.  (trw |nv'<length(x.u)∧nth(x.u,nv')=y| (open nth) * )
    ;NV'<LENGTH (X.U)∧NTH(X.U,NV')=Y

8.  (derive |∃n.n<length(x.u)∧nth(x.u,n)=y| * )
    (label m_n4)

9.  (assume |member(y,x.u)|)
    (label m_n5)

10. (rw * (open member))
    ;Y=X∨MEMBER(Y,U)

11. (cases * m_n3 m_n4)
    ;∃N.N<LENGTH (X.U)∧NTH(X.U,N)=Y

12. (ci m_n5)
    ;MEMBER(Y,X.U)⊃(∃N.N<LENGTH U'∧NTH(X.U,N)=Y)

13. (ci m_n1)

14. (ue (phi |λu.member(y,u)⊃(∃n.n<length u∧nth(u,n)=y)|) listinduction
       (open member nth) * )
    ;∀U.MEMBER(Y,U)⊃(∃N.N<LENGTH U∧NTH(U,N)=Y)
    (label member-nth)     ∎
```

8.10. Nthcdr.

```
(proof nthcdr)

1.  (decl nthcdr (syntype: constant) (type: |ground⊗ground→ground|))

2.  (defax nthcdr |∀x u n.nthcdr(nil,n)=nil∧nthcdr(u,0)=u∧
                         nthcdr(x.u,n')=nthcdr(u,n)|)
    (label simpinfo) (label nthcdrdef)

3.  (ue (phi3 |λu n.listp nthcdr(u,n)|) doubleinductionl)
    ;∀U N.LISTP NTHCDR(U,N)
    (label simpinfo)     ∎

4.  (ue (phi |λu.0<length u⊃nth(u,0).nthcdr(u,0')=u|) listinduction
```

```
        (part 2 (nuse nthdef)))
    ;∀U.0<LENGTH U⊃NTH(U,0).NTHCDR(U,1)=U
    (label nth_nthcdr_zero)    •

    ;car nthcdr

5.  (ue (phi3 |λu n.car(nthcdr(u,n))=nth(u,n)|) doubleinductionl)
    ;∀U N.CAR NTHCDR(U,N)=NTH(U,N)
    (label car-nthcdr)    ∎

    ;cdr nthcdr

6.  (ue (phi |λu.cdr(nthcdr(u,0))=nthcdr(u,0')|) listinduction)
    ;∀U.CDR U=NTHCDR(U,1)

7.  (ue (phi3 |λu n.cdr(nthcdr(u,n))=nthcdr(u,n')|) doubleinductionl * )
    ;∀U N.CDR NTHCDR(U,N)=NTHCDR(U,N')
    (label cdr_nthcdr)    ∎

    ;nthcdr car cdr

8.  (ue (phi |λu.0<length(u)⊃nthcdr(u,0)=nth(u,0).nthcdr(u,0')|)
        listinduction ( car-nthcdr cdr_nthcdr))

9.  (ue (phi3 |λu n.n<length(u)⊃nthcdr(u,n)=nth(u,n).nthcdr(u,n')|)
        doubleinductionl (use car-nthcdr cdr_nthcdr) * )
    ;∀U N.N<LENGTH U⊃NTHCDR(U,N)=NTH(U,N).NTHCDR(U,N')
    (label nthcdr-car-cdr) ∎

    ;nth in nthcdr

10. (ue (phi3 |λu n.∀m.n<m∧m<length u⊃member(nth(u,m),nthcdr(u,n))|)
doubleinductionl
        (use nthmember mode: exact)
(use proof-by-induction
     ue: ((a.|λm.(n'<m∧m<length(u)'⊃
                          member(nth(x.u,m),nthcdr(u,n)))|))
     mode: exact))
    ;∀U N M.N<M∧M<LENGTH U⊃MEMBER(NTH(U,M),NTHCDR(U,N))

11. (trw |∀u n m.n≤m∧m<length(u)⊃member(nth(u,m),nthcdr(u,n))|
         (open lesseq member)(use normal mode: always)
         (use * nthcdr-car-cdr mode: exact))
    ;∀U N M.N≤M∧M<LENGTH U⊃MEMBER(NTH(U,M),NTHCDR(U,N))
    (label nth_in_nthcdr) ∎

    ;nth nthcdr

12. (ue (phi3 |λu n.n<length u∧m<length(nthcdr(u,n))⊃
                    nth(nthcdr(u,n),m)=nth(u,m+n)|)
        doubleinductionl )
    ;∀U N.N<LENGTH U∧M<LENGTH (NTHCDR(U,N))⊃NTH(NTHCDR(U,N),M)=NTH(U,M+N)
    (label nth_nthcdr) ∎

    ;length nthcdr

13. (ue (phi3 |λu n.n≤length u⊃length(nthcdr(u,n))=length u-n|)
        doubleinductionl (use successor-minus mode: always)
        (open minus) (part 1#1#1 (open lesseq)))
    ;∀U N.N≤LENGTH U⊃LENGTH (NTHCDR(U,N))=LENGTH U-N
    (label length_nthcdr) ∎

    ;last nthcdr

14.(ue (phi |λu.nthcdr(u,length(u))=nil|) listinduction)
    ;∀U.NTHCDR(U,LENGTH U)=NIL
```

```
    (label last_nthcdr)    ■

    ;trivial nthcdr

15. (ue (phi3 |λu n.length(u)≤n⊃nthcdr(u,n)=nil|) doubleinductionl
        (part 1#1 (open lesseq)))
    (label trivial_nthcdr) ■

    ; allp,nthcdr

16. (ue (phi3 |λu n.allp(a,u)⊃allp(a,nthcdr(u,n))|) doubleinductionl
        (open allp) )
    ;∀U N.ALLP(A,U)⊃ALLP(A,NTHCDR(U,N))
    (label allp,nthcdr) ■
```

8.10.1. Nthcdr Induction.

Using induction on *n, we* show:

$$\forall n.\mathtt{phi(nthcdr(u,length(u)-n))}.$$

For n=0, nthcdr(u,length(u)-n) is **NIL,** and we have phi(nil).

Assume phi (nthcdr (u, length(u) -n)). Since subtraction is defined as a total function on nonnegative integers, we have for n≥length(u),

$$\mathtt{length(u)-n = 0 = length(u)-n'}.$$

so in this case the induction step is trivial.

If n< length(u), then

$$\mathtt{length(u)-n = (length(u)-n')'}$$

by elementary arithmetic and

$$\forall k.k<\mathtt{length(u)} \supset (\mathtt{phi(nthcdr(u,k'))}) \supset \mathtt{phi(nthcdr(u,k))})$$

is the inductive step of our principle. We can complete the induction step by letting k to be length(u) -n':

$$\mathtt{phi(nthcdr(u,length(u)-n))} \supset \mathtt{phi(nthcdr(u,length(u)-n'))}.$$

Finally it is convenient to write

$$\mathtt{nthcdr(u,k)}$$

as

$$\mathtt{nth(u,k).nthcdr(u,k')}$$

(using lemma *Nthcdr Car Cdr*).

```
(proof nthcdr_induction)

1.  (assume |∀n.n<length(u)∧phi(nthcdr(u,n'))⊃
               phi(nth(u,n).nthcdr(u,n'))|)
    (label n_i_1)
    ;deps: (N-I-1)

2.  (derive |∀n.n<length(u)⊃
              (phi(nthcdr(u,n'))⊃phi(nthcdr(u,n)))| *
          (use nthcdr_car_cdr mode: always))
    (label n-i-2)
    ;deps: (N-I-1)

    ; two cases

3.  (derive |length(u)≤n∨n<length(u)| trichotomy2)
    (label n-i-cases)

    ;one completely trivial

4.  (assume |length(u)≤n|)
    (label n-i-cl)
    ;deps: (N_I_C1)

5.  (trw |phi(nthcdr(u,length(u)-n))⊃
          phi(nthcdr(u,length(u)-n'))|
         (open minus pred)(use total-subtraction n-i-cl mode: always))
    (label n_i_case1)
    ;PHI(NTHCDR(U,LENGTH U-N))⊃PHI(NTHCDR(U,LENGTH U-N'))
    ;deps: (N_I_C1)

    ;the other quite trivial too...

6.  (assume |n<length(u)|)
    (label n_i_c2)
    ;deps: (6)

7.  (ue (n |length(u)-(n')|) n-i-2
        (use n_i_c2)(use minusfactll ue: ((n.|length(u)|)) )
        (use minusfact10 mode: exact direction: reverse>>
    (label n_i_case2)
    ;PHI(NTHCDR(U,LENGTH U-N))⊃PHI(NTHCDR(U,LENGTH U-N'))
    ;deps: (N-I-1 N_I_C2)

8.  (cases n-i-cases n_i_case1 n_i_case2)
    ;PHI(NTHCDR(U,LENGTH U-N))⊃PHI(NTHCDR(U,LENGTH U-N'))
    ;deps: (N-I-1)

9.  (ue (a |λn.phi(nthcdr(u,length(u)-n))|)
        proof-by-induction *
        (part 1 (use last-nthcdr mode: exact) (open minus)) )
    (label n-i-5)
    ;PHI(NIL)⊃(∀N.PHI(NTHCDR(U,LENGTH U-N)))
    ;deps: (N-I-1)

    ;cosmetics

10. (assume |phi(nil)|)
    (label n-i-6)
    ;deps: (10)

11. (derive |∀n.phi(nthcdr(u,length u-n>> | (n-i-5 n-i-6))
    ;deps: (N_I_1 N-I-6)

12. (ue (n |length u|) * )
    ;PHI(U)
```

```
    ;deps: (N-I-1 N-I-6)

13. (ci (n-i-6 n_i_1))
    ;PHI(NIL)∧(∀N.N<LENGTH U∧PHI(NTHCDR(U,N'))⊃PHI(NTH(U,N).NTHCDR(U,N')))⊃
    ;PHI(U)
    (label nthcdr_induction) ∎
```

8.11. Fstposition.

```
    ;facts about fstposition
    (proof fstpositionprop)

1.  (trw |∀k.¬null k'|)
    (label simpinfo)

2.  (ue (phi |λu.(null fstposition(u,y)⊃¬member(y,u))∧
                 (member(y,u)⊃natnum fstposition(u,y))∧
                 (null fstposition(u,y)∨natnum fstposition(u,y))|) listinduction
       (part 1 (open member fstposition) (use normal mode: always)))
    ;∀U.(NULL FSTPOSITION(U,Y)⊃¬MEMBER(Y,U))∧
    ;   (MEMBER(Y,U)⊃NATNUM(FSTPOSITION(U,Y)))∧
    ;   (NULL FSTPOSITION(U,Y)∨NATNUM(FSTPOSITION(U,Y)))
    (label simpinfo)(label posfacts)   ∎

3   (ue (phi |λu.∀y.sexp fstposition(u,y)|) listinduction
       (part 1 (open member fstposition) (use normal mode: always)))
    ;∀U Y.SEXP FSTPOSITION(U,Y)
    (label simpinfo)(label sortpos)   ∎

    ;pos_length

4   (ue (phi |λu.∀y.member(y,u)⊃fstposition(u,y)<length(u)|) listinduction
       (part 1 (open member fstposition) (use normal mode: always)))
    ;∀U Y.MEMBER(Y,U)⊃FSTPOSITION(U,Y)<LENGTH U
    (label pos_length)   ∎
```

8.11.1 Fstposition and Nth.

```
    ;lemmata nth-fstposition and fstposition-nth

    ;lemma nth-fstposition

1   (ue (phi |λu.∀n.member(n,u)⊃nth(u,fstposition(u,n))=n|) listinduction
       (use normal mode: always)
       (open member fstposition nth))
    ;∀U N.MEMBER(N,U)⊃NTH(U,FSTPOSITION(U,N))=N
    (label nth-fstposition) ∎

    (proof fstposition-nth)

1.  (ue (phi |λu.0<length u⊃fstposition(u,nth(u,0))=0|)
       listinduction (open fstposition nth member))
    ;∀U.0<LENGTH U⊃FSTPOSITION(U,CAR U)=0

2.  (derive |n<length u ∧ x=nth(u,n) ⊃ member(x,u)| (nthmember))

3.  (derive |uniqueness(x.u)∧n<length u⊃¬x=nth(u,n)| * (open uniqueness)>
```

```
4.  (ue (phi3 |λu n.uniqueness u∧n<length u⊃fstposition(u,nth(u,n))=n|)
        doubleinduction1 *
        (open fstposition nth member uniqueness) -3 nthmember)
    ;∀U N.UNIQUENESS(U)∧N<LENGTH U⊃FSTPOSITION(U,NTH(U,N))=N
    (label fstposition_nth)      ■
```

8.12. Injectivity.

```
;injectivity
;another predicate for uniqueness

(proof inj)

1.  (decl (inj) (type: |ground→truthval|))
2.  (define inj |∀u.inj(u)=∀n m.n<length(u)∧m<length(u)∧nth(u,n)=nth(u,m)⊃n=m|)
    (label injdef)
```

We want to show that the following properties of a list, u are equivalent:

(i) *uniqueness*: for every member x, x does not belong to the tail of u after x;

(ii) *injectivity*: if nth(u,i) = nth(u,j) then i = j.

The property of uniqueness holds for all the tails of a list, if it holds for the list: this is fact (needed later, line 1-4) is easily established by double induction on lists and numbers.

```
;equivalence of uniqueness and inj

(proof uniqueness_inj)

1.  (ue (phi3 |λu n.uniqueness u⊃uniqueness nthcdr(u,n)|) doubleinduction1
            (open uniqueness nthcdr))
    ;∀U N.UNIQUENESS(U)⊃UNIQUENESS(NTHCDR(U,N))
    (label uniqueness_nthcdr)
```

Assume uniqueness(u) (line 2). We want to show inj (u). Therefore we assume nth(u,i) = nth(u, j), with i and j both less than length(u) (lines 2, 3 and 4). We need to obtain i = j (line 13). We will derive **a** contradiction from the assumption that either i < j or j < i (lines 9 and 12) and apply the trichotomy:

$$\forall n\ m.m<n \lor m=n \lor n<m.$$

Assume i < j. Then nth(u, j) is a member of nthcdr u,i') (line 8). (this is the fact *Nth in Nthcdr*). But this contradicts the fact that nthcdr(u,n) enjoys the *uniqueness* property. So $\neg n < m$.

Similarly for m < n.

```
2.  (assume|uniqueness(u)|)
    (label ui1)

3.  (assume I i<length u|)
     label ui2)

4.  (assume |j<length u|)
    (label ui3)

5.  (assume|nth(u,i)=nth(u,j)|)
    (label ui4)

6.  (derive|uniqueness(nthcdr(u,i))|(uniqueness_nthcdr ui1))
```

```
7.  (rw * (use nthcdr_car_cdr ui2 mode: always) (open uniqueness))
    ;¬MEMBER(NTH(U,I),NTHCDR(U,I'))∧UNIQUENESS(NTHCDR(U,I'))
    ;deps: (UI1 UI2)

    ;labels: NTH_IN_NTHCDR
    ;∀U N M.N≤M∧M<LENGTH U⊃MEMBER(NTH(U,M),NTHCDR(U,N))

8.  (ue ((u.u)(n.|i'|)(m.j)) nth_in_nthcdr
        (use ui4 mode: exact direction: reverse>
        (use ui3 * mode: exact))
    ;¬I'≤J
    ;deps: (UI1 U12 U13 UI4)

    ;labels: LESS-LESSEQSUCC
    ;∀M N.M<N=M'≤N

9.  (ue ((m.i)(n.j)) less-lesseqsucc * )
    ;¬I<J
    (label ui_way1)
    ;deps: (UI1 U12 U13 UI4)

10. (ci (ui1 ui2 ui3 ui4))
    ;UNIQUENESS(U)∧I<LENGTH U∧J<LENGTH U∧NTH(U,I)=NTH(U,J)⊃¬I<J

11. (ue ((i.j)(j.i)) * )
    ;UNIQUENESS(U)∧J<LENGTH U∧I<LENGTH U∧NTH(U,J)=NTH(U,I)⊃¬J<I

12. (derive |¬j<i| (* uil ui2 ui3 ui4))
    (label ui_way2)
    ;deps: (UI1 U12 U13 UI4)

13. (derive |i=j| (trichotomy ui_way1 ui_way2))
    ;deps: (UI1 U12 U13 UI4)

14. (ci (ui1 ui2 ui3 ui4))
    ;UNIQUENESS(U)∧I<LENGTH U∧J<LENGTH U∧NTH(U,I)=NTH(U,J)⊃I=J

15. (trw |uniqueness(u)⊃inj(u)| * (open inj))
    ;UNIQUENESS(U)⊃INJ(U)
    (label uniqueness_inj)
```

We prove inj (u)⊃uniqueness(u) by listinduction. It is easy to see that inj (x . u) implies inj (u) (line 4) and hence uniqueness(u), by induction hypothesis. We need to show ¬member (x , u) . in order to conclude uniqueness (x . u). If x was a. member of u. it would be the $(n_0 + 1)$-th member of $x.u$, for some n_0, and we would have nth(x.u,$(n_0 + 1)$)=nth(x.u,0) (line 7): by the definition of inj this implies $n_0 + 1 = 0$.

```
1.  (assume |inj(u)⊃uniqueness(u)|)
    (label inj_un1)

2.  (assume |inj(x.u) I>
    (label inj_un2)

3.  (rw * (open inj))
    ;∀N M.N<LENGTH U'∧M<LENGTH U'∧NTH(X.U,N)=NTH(X.U,M)⊃N=M
    (label inj_un3)
    ;deps: (INJ_UN2)

4.  (trw linj ul (open inj) (use * ue: ((n.|n'|)(m.|m'|))))
    ;INJ(U)

5.  (derive Iuniqueness u| (* inj_un1))
    (label inj_un4)
```

```
;deps: (INJ_UN2)

6.  (assume |member(x,u)|)
    (label inj_un5)

7.  (define nv |nv'<length(x.u)∧nth(x.u,nv')=nth(x.u,0)|
          (* member_nth))
    ;NV is unknown.
    ;the symbol NV is given the same declaration as N
    ;deps: (INJ_UN5)

8.  (rw *)
    ;NV<LENGTH U∧NTH(U,NV)=X
    ;deps: (INJ_UN5)

9.  (ue ((n.|nv'|)(m.|0|)) inj_un3 *)
    ;FALSE
    ;deps: (INJ_UN2 INJ_UN5)

10. (ci inj_un5)
    ;¬MEMBER(X,U)
    ;deps: (INJ_UN2)

11. (trw |uniqueness(x.u)|(open uniqueness) (* inj_un4))
    11. ;UNIQUENESS(X.U)
    ;deps: (INJ_UN1 INJ_UN2)

12. (ci inj_un2)
    ;INJ(X.U)⊃UNIQUENESS(X.U)
    ;deps: (INJ_UN1)

13. (ci INJ_UN1)
    ;(INJ(U)⊃UNIQUENESS(U))⊃(INJ(X.U)⊃UNIQUENESS(X.U))

14. (ue (phi |λu.inj(u)⊃uniqueness(u)|) listinduction
       * (part 1#1 (open inj uniqueness)))
    ;∀U.INJ(U)⊃UNIQUENESS(U)
    (label inj_uniqueness)

15. (derive|∀u.uniqueness(u ≡inj(u)|(uniqueness-inj inj_uniqueness))
    (label  uniqueness-inject vity)     ■
```

8.13. Nth, Allp and Mklset.

```
;proof of facts about sets

(proof setfacts)
;nth_allp

1.  (assume|∀n.n<length(u)⊃phi1(nth(u,n))|)
    (label allp_intr1)

2.  (ue ((phi.|λu.allp(phi1,u)|)(u.u)) nthcdr_induction
       (open allp) (use * mode: always))
    ;ALLP(PHI1,U)

3.  (ci allp_intr1)
    ;(∀N.N<LENGTH U⊃PHI1(NTH(U,N)))⊃ALLP(PHI1,U)
    (label  nth_allp) ■

;mklset_fact
```

```
4.  (derive |∀x.(mklset(u))(x)≡(∃k.k<length u∧nth(u,k)=x)|
           (nthmember member_nth) (open mklset))

5.  (ue ((av.|mklset(u)|)(bv.|λx.(∃k.k<length u∧nth(u,k)=x)|)) set-extensionality
       * (open epsilon) )
    ;MKLSET(U)=(λX.(∃K.K<LENGTH U∧NTH(U,K)=X))
    (label mklset_fact )      ■

    (save-proofs nth)
```

8.14. file APPL: Functions Represented by Association Lists.

```
    ;function as alists: the notion of application for association lists

    (proof appalist)

1.  (decl dom (type: |ground→ground|))
2.  (defax dom |∀xa y alist.dom nil=nil∧
                              dom((xa.y).alist)=xa.dom alist|
    (label domdef)

3.  (decl range (type: |ground→ground|))
4.  (defax range |∀xay alist.range nil=nil∧
                                range((xa.y).alist)=y.range alist |)
    (label rangedef )

5.  (decl functp (type: |ground→truthval|))
6.  (define functp |∀alist.functp(alist)≡uniqueness dom(alist)|)
    (label functdef)

7.  (decl injectp (type: |ground→truthval|))
8.  (define injectp |∀alist.injectp(alist)≡functp(alist)∧uniqueness range(alist)|)
    (label injectdef )

9.  (decl (appalist) (type: |ground⊗ground→ground|))
10. (define appalist |∀alisty.appalist(y,alist)=cdr assoc(y,alist)|)
    (label appalistdef)

11. (decl (samemap) (type: |ground⊗ground→truthval|))
12. (define samemap
     |∀alist alist1.samemap(alist,alist1)≡
                 mklset dom(alist)=mklset dom(alist1)∧
                 (∀y.y∈mklset dom(alist)⊃appalist(y,alist)=appalist(y,alist1)) |)
    (label samemapdef )

13. (define permutp |∀alist.permutp(alist)≡
                        functp(alist)∧mklset(dom(alist))=mklset(range(alist))|)
    (label permutp_def)
```

8.14.1. Alist Induction.

```
(proof alistind)

1.  (assume |chi(nil)∧(∀xa y alist.chi(alist)⊃chi((xa.y).alist))|).
    (label alind1)

2.  (assume |alistp u⊃chi u|)
    (label alind2)

3.  (assume |alistp (x.u)|)
    (label alind3)

4.  (ue (alist |x.u|) alistdef1 * )
    ;¬ATOM X∧ATOM CAR X∧ALISTP U

5.  (derive |(∀xa y alist.chi(alist)⊃chi((xa.y).alist))| alind1)

6.  (ue ((xa.|car x|)(y.|cdr(x)|)(alist.u)) * -2 alind3 alind2)
    ;CHI(X.U)
    ;deps: (ALIND1 ALIND2 ALIND3)

7.  (ci alind3)
    ;ALISTP X.U⊃CHI(X.U)
    ;deps: (ALIND1 ALIND2)

8.  (ci alind2)
    ;(ALISTP U⊃CHI(U))⊃(ALISTP X.U⊃CHI(X.U))
    ;deps: (ALIND1)

9.  (ue (phi |λu.alistp(u)⊃chi(u)|) listinduction * alind1)
    ;∀U.ALISTP U⊃CHI(U)

10  (derive |∀alist.chi alist| * )
    ;deps: (ALIND1)

11  (ci alind1)
    ;CHI(NIL)∧(∀XA Y ALIST.CHI(ALIST)⊃CHI((XA.Y).ALIST))⊃(∀ALIST.CHI(ALIST))
    (label alistinduction) ■
```

8.14.2. Facts About Association Lists.

```
    ;facts about alists
    (proof alistfacts)

    ;domsort

1.  (ue (chi |λalist.listp dom(alist)|) alistinduction (open dom))
    ;∀ALIST.LISTP DOM(ALIST)
    (label simpinfo)(label domsort)    ■

2.  (ue (chi |λalist.listp range(alist)|) alistinduction (open range))
    ;∀ALIST.LISTP RANGE(ALIST)
    (label simpinfo)(label rangesort)    ■

    ;domlength

3.  (ue (chi |λalist.length dom alist=length alist|) alistinduction (open dom))
    ;∀ALIST.LENGTH (DOM(ALIST))=LENGTH ALIST
    (label domlength)    ■
```

```
;domrangelength

4.  (ue (chi |λalist.length(dom alist)=length(range alist)|) alistinduction
        (open dom range>)
    ;∀ALIST.LENGTH (DOM(ALIST))=LENGTH (RANGE(ALIST))
    (label domrangelength) ■

;appalistsort

5.  (ue (chi |λalist.sexp appalist(y,alist)|) alistinduction
        (part l(open appalist assoc)))
    ;∀ALIST.SEXP APPALIST(Y,ALIST)
    (label simpinfo)(label appalistsort)    ■

;trivial appalist

6.  (ue (chi|λalist.¬(y∈mklset dom(alist))⊃appalist(y,alist)=nil|) alistinduction
        (part 1 (open epsilon mklset dom appalist assoc member>>>
    ;∀ALIST.¬Y∈MKLSET(DOM(ALIST))⊃APPALIST(Y,ALIST)=NIL
    (label trivial-appalist) ■
```

8.14.3. Samemap Definition.

```
(proof samemap)

1.  (trw |samemap(alist,alist)|(open samemap))
    ;SAMEMAP(ALIST,ALIST)
    (label  samemap-equivalence)

2.  (tra |samemap(alist,alist1)⊃samemap(alist1,alist)| (open samemap mklset dom))
    ;SAMEMAP(ALIST,ALIST1)⊃SAMEMAP(ALIST1,ALIST)
    (label  samemap-equivalence)

3.  (trw |samemap(alist,alist1)∧samemap(alist1,alist2)⊃samemap(alist,alist2)|
        (open samemap mklset dom))
    ;SAMEMAP(ALIST,ALIST1)∧SAMEMAP(ALIST1,ALIST2)⊃SAMEMAP(ALIST,ALIST2)
    (label  samemap-equivalence)     ■


;apparently stronger definition of samemap
(proof samemapdef)

1.  (assume|samemap(alist1,alist2)|)

2.  (ra * (open samemap))
    ;MKLSET(DOM(ALIST1))=MKLSET(DOM(ALIST2))∧
    ;(∀Y.Y∈MKLSET(DOM(ALIST1))⊃APPALIST(Y,ALIST1)=APPALIST(Y,ALIST2))

3.  (tra |¬y∈mklset(dom(alist1))⊃appalist(y,alist1)=appalist(y,alist2)|
         (use trivial,appalist mode: always)
         (use * mode: exact))
    ;¬Y∈MKLSET(DOM(ALIST1))⊃APPALIST(Y,ALIST1)=APPALIST(Y,ALIST2)

4.  (ue ((q.|y∈mklset dom(alist1)|)(p.|appalist(y,alist1)=appalist(y,alist2)|))
        excluded-middle * -2)
    ;APPALIST(Y,ALIST1)=APPALIST(Y,ALIST2)

5.  (derive|mklset(dom(alist1))=mklset(dom(alist2))∧
            ∀y.appalist(y,alist1)=appalist(y,alist2)| (-3 *))

6.  (ci -5)
    ;SAMEMAP(ALIST1,ALIST2)⊃
```

```
;MKLSET(DOM(ALIST1))=MKLSET(DOM(ALIST2))∧
;(∀Y.APPALIST(Y,ALIST1)=APPALIST(Y,ALIST2))
```

```
7.  (derive|samemap(alist1,alist2)≡
            (mklset(dom(alist1))=mklset(dom(alist2))∧
            (∀x.appalist(x,alist1)=appalist(x,alist2)))| * )
    (label  samemap_def1) ■
```

8.15. Functions Represented by Lists of Numbers.

```
;functions as lists of numbers

(wipe-out)
(get-proofs  pigeon)

(proof appl)

1   (define appl |∀u i.appl(u,i)=nth(u,i)|)
    (label appldef)

2   (axiom |∀u i.i<length u ⊃ sexp(appl(u,i))∧member(appl(u,i),u)|)
    (label applfacts) (label simpinfo)

;predicates for functions

3   (decl (into) (type: |ground→truthval|))
4   (define into |∀u.into(u)=(∀n.n<length u⊃natnum nth(u,n)∧nth(u,n)<length u)|>
    (label intodef)

5   (decl (onto) (type: |ground→truthval|))
6   (define onto |∀u.onto(u)=(into(u)∧(∀n.n<length u⊃member(n,u)))|)
    (label ontodef)

7   (decl (perm) (type: |ground→truthval|))
8   (define perm |∀u.perm(u)=onto(u)|)

;injectivity is given by thepredicate inj

(save-proofs appl)
```

8.15.1. Extensionality.

```
(wipe-out)
(get-proofs appl)

(proof  extensionality)

1.  (assume |length u=length v∧(∀i.i<length v⊃nth(u,i)=nth(v,i))⊃u=v|)
    (label ext1)

2.  (assume llength u=length v|)
    (label ext2)

3.  (assume |∀i.i<length v'⊃nth(x.u,i)=nth(y.v,i)|)
    (label ext3)

4   (ue (i 0) * ext2)
```

```
     ;X=Y
     (label ext4)
     ;deps: (EXT2 EXT3)

5.   (ue (i |i'|) ext3 ext2)
     ;I<LENGTH V⊃NTH(U,I)=NTH(V,I)
     (label ext5)
     ;deps: (EXT2 EXT3)

6.   (derive |u=v|(ext1 ext2 ext5))
     (label ext6)
     ;deps: (EXT1 EXT2 EXT3)

7.   (trw |x.u=y.v| (use ext4 ext6 mode: exact))
     ;X.U=Y.V
     ;deps: (EXT1 EXT2 EXT3)

8.   (ci (ext2 ext3))
     ;LENGTH U=LENGTH V∧(∀I.I<LENGTH U'⊃NTH(X.U,I)=NTH(Y.V,I))⊃X.U=Y.V
     ;deps: (EXT1)

9.   (ci ext1)
     ;(LENGTH U=LENGTH V∧(∀I.I<LENGTH U⊃NTH(U,I)=NTH(V,I))⊃U=V)⊃
     ;(LENGTH U=LENGTH V∧(∀I.I<LENGTH U'⊃NTH(X.U,I)=NTH(Y.V,I))⊃X.U=Y.V)

10.  (ue (phi2 |λu v.length u=length v∧(∀i.i<length u⊃nth(u,i)=nth(v,i))⊃u=v|)
        doubleinduction (open nth) * )
     ;∀U V.LENGTH U=LENGTH V∧(∀I.I<LENGTH V⊃NTH(U,I)=NTH(V,I))⊃U=V
     (label extensionality) ■

11.  (trw |∀u i.i<length u ⊃ sexp(appl(u,i))∧member(appl(u,i),u) |
       (open appl) nthmember)
     ;∀U I.I<LENGTH U⊃SEXP APPL(U,I)∧MEMBER(APPL(U,I),U)
     (label applfact) (label simpinfo)     ■
```

8.16. file SUMS: Finite Union and Finite Sum.

```
     ;the notions of finite union and finite sum
     (wipe-out)
     (get-proofs appl)
     (proof sums>

1.   (decl allnum (type: |ground⊗@set→truthval|)(syntype: constant))
2.   (decl somenum (type: |ground⊗@set→truthval|)(syntype: constant))
3.   (decl (numseq f) (type:|ground→ground|))
4.   (decl sum (type: |(@numseq)⊗(@n)→(@n)|)(syntype: constant))
5.   (decl setseq (type: |@n→@set|))
6.   (decl un (type: |(@setseq)⊗(@n)→(@set)|)(syntype: constant))

     ;axiom for allnum
7.   (defax allnum|∀n a.allnum(0,a)∧(allnum(n',a)≡a(n)∧allnum(n,a))|)
     (label allnumdef)

     ;axiom for somenum
8.   (defax somenum|∀n a.¬somenum(0,a)∧(somenum(n',a)≡a(n)∨somenum(n,a))|)
     (label somenumdef)

     ;axiom for sum
9.   (defax sum|∀n numseq.sum(numseq,0)=0∧sum(numseq,n')=sum(numseq,n)+numseq(n)|)
     (label sumdef)

     ;axiom for un
```

```
10. (defax un |∀n setseq.un(setseq,0)=emptyset∧un(setseq,n')=un(setseq,n)∪setseq(n)|)
    (label undef)

11. (decl disj-pair (syntype: constant) (type: |(@set⊗@set)→truthval|))
12. (define disj-pair |∀a b.disj_pair(a,b)=emptyp(a∩b)|)
    (label disjpair_def)

13. (decl disjoint (syntype: constant) (type: |((ground→@set)⊗ground)→truthval|))
14. (defax disjoint |∀n setseq.disjoint(setseq,0)∧
       disjoint(setseq,n')=(disjoint(setseq,n)∧disj_pair(un(setseq,n),setseq(n)))| )
    (label disjointdef)
```

8.16.1. Bound Quantifiers.

```
(proof allnumprop)

;we can easily prove that 'allnum' does its job

1.  (ue (a |λn.allnum(n,a)⊃(m<n⊃a(m))|) proof-by-induction
       (use transitivity-of-order) (use successor11 (open allnum)
       (use less-succ-lesseq normal mode: exact) (open lesseq))
    ;∀N.ALLNUM(N,A)⊃(M<N⊃A(M))

2.  (ue (a |λn.(∀m.m<n⊃a(m))⊃allnum(n,a)|) proof-by-induction
       (open allnum) (use normal mode: always)
       (use less-succ-lesseq mode: exact) (open lesseq))
    ;∀N.(∀M.M<N⊃A(M))⊃ALLNUM(N,A)

3.  (derive |∀n.(∀m.m<n⊃a(m))≡allnum(n,a) I (* -2))

    ;similarly for 'somenum':

4.  (ue (a |λn.somenum(n,a)⊃(∃m.m<n∧a(m))|) proof-by-induction
       (use transitivity-of-order) (use successor1) (open somenum)
       (part 1 (der))
       (use less-succ-lesseq normal mode: exact) (open lesseq))
    ;∀N.SOMENUM(N,A)⊃(∃M.M<N∧A(M))

5.  (ue (a |λn.(∃m.m<n∧a(m))⊃somenum(n,a)|) proof-by-induction
       (open somenum) (use normal mode: always) (part 1(der))
         (use less-succ-lesseq mode: exact) (open lesseq))
    ;∀N.(∃M.M<N∧A(M))⊃SOMENUM(N,A)

6.  (derive |∀n.(∃m.m<n∧a(m))≡somenum(n,a)| (* -2))
```

8.16.2. Facts About Sums and Unions.

```
(proof unionprop)

;a property of union

;unionfact1

(ue (a |λn.m<n⊃(∀xv.(setseq(m))(xv)⊃(un(setseq,n))(xv))|)
    proof-by-induction
    (open un union) (use less-succ-lesseq mode: always)
    (open lesseq) (use normal mode: always))
;∀N.M<N⊃(∀XV.(SETSEQ(M))(XV)⊃(UN(SETSEQ,N))(XV))
```

```
;namely:

2.  (trw |∀setseq n m.m<n⊃setseq(m)⊂un(setseq,n)| * (open inclusion))
    (label unionfact1) ■

    ;a property of sum

    ;sumsort

3.  (ue (a |λn.allnum(n,λm.natnum numseq(m))⊃natnum sum(numseq,n)|)
        proof-by-induction (open allnum sum>>
    ;∀N.ALLNUM(N,λM.NATNUM(NUMSEQ(M)))⊃NATNUM(SUM(NUMSEQ,N))

4.  (rw * (use allnumfact mode: exact direction: reverse>>
    ;∀N.(∀M.M<N⊃NATNUM(NUMSEQ(M)))⊃NATNUM(SUM(NUMSEQ,N))
    (label sumsort) ■

    ;mksetfact

5.  (ue (a |λn.n≤length u⊃
            (un(λm.mkset(nth(u,m)),n))(x)≡somenum(n,λk.x=nth(u,k))|)
        proof-by-induction
        (part 1(open un mkset nth somenum union emptyset) (der))
        (use succ_lesseq_lesseq mode: always))
    ;∀N.N≤LENGTH U⊃(UN(λM.MKSET(NTH(U,M)),N))(X)≡SOMENUM(N,λK.X=NTH(U,K))

6.  (rw * (use somenumfact
          ue: ((a.|λk.x=nth(u,k)|)(n.n)) mode: exact direction: reverse>>
    ;∀N.N≤LENGTH U⊃(UN(λM.MKSET(NTH(U,M)),N))(X)≡(∃M.M<N∧X=NTH(U,M))

7.  (assume |n≤length u|)

8.  (ue ((av.|un(λm.mkset nth(u,m),n)|)
         (bv.|λx.∃k.k<n∧nth(u,k)=x|))set-extensionality
         (open epsilon)(use * -2 mode: always))
    ;UN(λM.MKSET(NTH(U,M)),N)=(λX.(∃K.K<N∧NTH(U,K)=X))

9.  (ci -2)
    ;N≤LENGTH U⊃UN(λM.MKSET(NTH(U,M)),N)=(λX.(∃K.K<N∧NTH(U,K)=X))
    (label mksetfact) ■

    ;mklset_un

10. (ue (n llength u|)mksetfact
        (use mklset_fact mode: exact direction: reverse> (open lesseq))
    ;∀U.UN(λM.MKSET(NTH(U,M)),LENGTH U)=MKLSET(U)
    (label mklset_un) ■
```

8.17. file MULT: Multiplicity.

```
;the notion of multiplicity
(wipe-out)
(get-proofs sums>
(proof multiplicity)

1. (decl mult (type: |(ground⊗@set)→ground|))
2. (defax mult |∀x u a.mult(nil,a)=0∧
                   mult(x.u,a)=if a(x) then mult(u,a)' else mult(u,a) |>
   (label mult_def)

   ;facts about multiplicity

3. (ue (phi |λu.∀a.natnum(mult(u,a))|) listinduction
       (use mult-def mode: always))
   (label simpinfo) (label multfact)   ■

   ;multiplicity is less or equal to length

4. (ue (phi |λu.mult(u,a)≤length(u)|) listinduction
       lesseq_lesseq_succ (open mult length) (part 1#1(open lesseq)))
   ;∀U.MULT(U,A)≤LENGTH U
   (label  length_mult) ■

   ;if there is a member, multiplicity is not zero

5. (ue (phi |λu.∀y a.member(y,u)∧a(y)⊃0<mult(u,a)|) listinduction
       (open mult member) (use normal mode: always))
   ;∀U Y A.MEMBER(Y,U)∧A(Y)⊃0<MULT(U,A)

6, (rw * use less-lesseqsucc mode: always))
   ;∀U Y A.MEMBER(Y,U)∧A(Y)⊃1≤MULT(U,A)
   (label  member_mult) ■

   ;multiplicity of the emptyset

7. (ue (phi |λu.mult(u,emptyset)=0|) listinduction
       (part 1(open emptyset mult)))
   ;∀U.MULT(U,EMPTYSET)=0
   (label simpinfo) (label emptyfacts)   ■

   ;mult_nthcdr

   ;we prepare a rewriter

8. (ue ((q.|mult(nthcdr(u,n'),a)'≤mult(u,a) |>
        (r.|mult(nthcdr(u,n'),a)≤mult(u,a)|)
        (p.|a(nth(u,n))|)) trans_cond
       (use succ_lesseq_lesseq ue: ((m.|mult(nthcdr(u,n'),a)|)
                                    (n.|mult(u,a)|)) mode: exact ))
   ;(IF A(NTH(U,N)) THEN MULT(NTHCDR(U,N'),A)'≤MULT(U,A)
   ;    ELSE MULT(NTHCDR(U,N'),A)≤MULT(U,A))⊃MULT(NTHCDR(U,N'),A)≤MULT(U,A)

   ;conclusion

9. (ue (a |λn.∀a u.n<length(u)⊃mult(nthcdr(u,n),a)≤mult(u,a)|) proof-by-induction
       (part 1#1 (open lesseq)) succ_less_less
       (part 1#2#1#1 (use nthcdr_car_cdr mode: always))
       (open mult) * )
   ;∀N A U.N<LENGTH U⊃MULT(NTHCDR(U,N),A)≤MULT(U,A)
   (label mult_nthcdr)   ■
```

8.17.1. Multiplicity Implies Injectivity.

```
(proof multinj_computation)

;a sublemma to compute multiplicity

1.  (assume |j<length v|)
    (label mc0)

2.  (assume |i<j|)
    (label mc1)

3.  (assume|nth(v,i)=nth(v,j)|)
    (label mc2)

4.  (derive |i<length v|(mc0 mcl transitivity-of-order))
    (label mc3)
    ;deps: (mc0 mc1)

    ;labels: NTH_IN_NTHCDR
    ;∀U N M.N≤M∧M<LENGTH U⊃MEMBER(NTH(U,M),NTHCDR(U,N))

5.  (ue ((u.v)(n.|i'|)(m.j)) nth_in_nthcdr mc0 mcl
        (use less-lesseqsucc mode: exact direction: reverse>>
    ;MEMBER(NTH(V,J),NTHCDR(V,I'))
    (label mc4)
    ;deps: (MC0 MC1)

    ;labels: MEMBER_MULT
    ;∀U Y A.MEMBER(Y,U)∧A(Y)⊃1≤MULT(U,A)

6.  (ue ((u.|nthcdr(v,i')|)(y.|nth(v,j)|)(a.|mkset nth(v,j)|)) member-mult
        (part 1(open lesseq mkset)) mc4
        (use mc2 mode: exact direction: reverse>>
    ;1≤MULT(NTHCDR(V,I'),MKSET(NTH(V,I)))
    (label mc5)
    ;deps: (MC0 MC1 MC2)

7.  (trw |n≤mult(nthcdr(v,i'),mkset nth(v,i))⊃n'≤mult(nthcdr(v,i),mkset nth(v,i))|
         (open mult mkset)(use nthcdr_car_cdr mc3 mode: exact))
    ;N≤MULT(NTHCDR(V,I'),MKSET(NTH(V,I)))⊃N'≤MULT(NTHCDR(V,I),MKSET(NTH(V,I)))
    ;deps: (MC0 MC1)

8.  (ue (n |1 |) * mc5)
    ;2≤MULT(NTHCDR(V,I),MKSET(NTH(V,I)))
    (label mc6)
    ;deps: (MC0 MC1 MC2)

    ;labels: MULT_NTHCDR
    ;∀A U N.N<LENGTH U⊃MULT(NTHCDR(U,N),A)≤MULT(U,A)

9.  (ue ((n.i)(u.v)(a.|mkset nth(v,i) I>> mult_nthcdr mc3)
    ;MULT(NTHCDR(V,I),MKSET(NTH(V,I)))≤MULT(V,MKSET(NTH(V,I)))
    ;deps: (MC0 MC1)

    ;labels: TRANS_LESSEQ
    ;∀N M K.N≤M∧M≤K⊃N≤K

10. (ue ((n.|2|)(m.|mult(nthcdr(v,i),mkset nth(v,i))|)(k.|mult(v,mkset nth(v,i))|))
        trans_lesseq mc6 * )
    ;2≤MULT(V,MKSET(NTH(V,I)))
    ;deps: (MC0 MC1 MC2)

11. (ci (mc1 mc0 mc2))
    ;I<J∧J<LENGTH V∧NTH(V,I)=NTH(V,J)⊃2≤MULT(V,MKSET(NTH(V,I)))
```

```
    (label  multinj-computation)   ■

    ;lemma multiplicity implies injectivity
    (proof mult_inj)

1.  (assume |∀k.k<length v⊃mult(v,mkset(nth(v,k)))=1|)
    (label mi1)

2.  (assume |i<length v∧j<length v∧nth(v,i)=nth(v,j)|)
    (label mi2)

3.  (ue ((v.v)(i.i)(j.j)) multinj-computation mi2
        (use mil ue: ((k.i))mode: exact)(open lesseq))
    ;¬I<J
    ;deps: (MI1 MI2)

4.  (ue ((v.v)(i.j)(j.i)) multinj-computation mi2
        (use mil ue: ((k.j)) mode: exact)(open lesseq))
    ;¬J<I
    ;deps: (MI1 MI2)

5.  (derive |i=j|(trichotomy * -2))
    ;deps: (MI1MI2)

6.  (ci mi2)
    ;I<LENGTH V∧J<LENGTH V∧NTH(V,I)=NTH(V,J)⊃I=J
    ;deps: (MI1)

7.  (trw |inj v| (open inj) * )
    ;INJ(V)
    ;deps: (MI1)

8.  (ci mi1)
    ;(∀K.K<LENGTH V⊃MULT(V,MKSET(NTH(V,K)))=1)⊃INJ(V)
    (label  mult_inj)  ■
```

8.17.2. The Multiplicity of Union is the Sum of Multiplicities.

```
;Lemma:if the union is disjoint, then the multiplicity of the union is
;the sum of the multiplicities

    (proof multsum)

1.  (ue (phi |λu. disj_pair(a,b)⊃mult(u,a∪b)=mult(u,a)+mult(u,b)|)
        listinduction
        (part 1 (open mult union disj_pair emptyp intersection)
                (use normal mode: always))
        (part 1 (der)) )
    ;∀U.DISJ_PAIR(A,B)⊃MULT(U,A∪B)=MULT(U,A)+MULT(U,B)
    (label  multsum) ■

2.  (ue (a |λn.disjoint(setseq,n)⊃
              mult(u,un(setseq,n))=sum(λx1.mult(u,setseq(x1)),n)|)
        proof-by-induction (open disjoint un sum mult ) multfact
        (use multsum mode: exact) (use normal mode: always))
    ;∀N.DISJOINT(SETSEQ,N)⊃MULT(U,UN(SETSEQ,N))=SUM(λX1.MULT(U,SETSEQ(X1)),N)
    (label  mult_of_un_is_sum_mult) ■
```

8.18. file PIGEON: the Pigeon Hole Principle in II Order Arithmetic,

```
     (wipe-out)
     (get-proofs sums>
     (proof pigeonfact)

1.   (assume |∀n.natnum f(n)|)
     (label sort1

2.   (ue ((numseq.|λk.f(k)|)(n.n)) sumsort * )
     ;NATNUM(SUM(λK.F(K),N))
     (label sort2)

3.   (ue (a |λn.allnum(n,λk.1≤f(k))⊃n≤sum(λk.f(k),n)|)
        proof-by-induction
        (open allnum sum>  zeroleast (use sort1 sort2 mode: always)
        (use add-lesseq ue: ((n.n)(k.|f(n)|)(m.|sum(λk.f(k),n)|)) ))
     (label  strictly-increasing)
     ;∀N.ALLNUM(N,λK.1≤F(K))⊃N≤SUM(λK.F(K),N)
     ;deps: (SORT1)

4.   (ue (a |λn.allnum(n,λk.1≤f(k))∧sum(λk.f(k),n)=n⊃allnum(n,λk.1=f(k))|)
        proof-by-induction
        (open allnum sum> strictly-increasing sort1 sort2
        (use add-one
            ue: ((k.|f(n)|)(n.n)(m.|sum(λk.f(k),n)|)) mode: always))
     ;∀N.ALLNUM(N,λK.1≤F(K))∧SUM(λK.F(K),N)=N⊃ALLNUM(N,λK.1=F(K))

     ;in more conventional notation:

5.   (rw * (use allnumfact ue: ((a.|λk.1≤f(k)|)(n.n))
               mode: always direction: reverse>
          (use allnumfact ue: ((a.|λk.1=f(k)|)(n.n))
               mode: always direction: reverse)>
     ;∀N.(∀M.M<N⊃1≤F(M))∧SUM(λK.F(K),N)=N⊃(∀M.M<N⊃1=F(M))
     ;deps: (SORT1)

6    (ci sort1)
     ;(∀N.NATNUM(F(N)))⊃
     ;(∀N.(∀M.M<N⊃1≤F(M))∧SUM(λK.F(K),N)=N⊃(∀M.M<N⊃1=F(M)))

     ;application to lists

     (proof pigeonlist)

1    (assume |disjoint(setseq,length u) |>
     (label pl1)

     ;multiplicity less than length

-2   (ue ((u.u)(a.|un(setseq,length  u)|)) length_mult)
     ;MULT(U,UN(SETSEQ,LENGTH U))≤LENGTH U
     (label pl2)

3    (derive |sum(λm.mult(u,setseq(m)),length u)≤length u|
            (mult_of_un_is_sum_mult pl1 pl2))
     (label pl3)

4.   (ue ((f.|λm.mult(u,setseq(m))|)(n.|length u|))pigeonfact pl3 multfact)
     ;(∀M.M<LENGTH U⊃1≤MULT(U,SETSEQ(M)))⊃(∀M.M<LENGTH U⊃1=MULT(U,SETSEQ(M)))
     ;deps: (PL1)

     ;the pigeon hole principle on lists

5.   (ci pl1)
```

```
;DISJOINT(SETSEQ,LENGTH U)⊃
;((∀M.M<LENGTH U⊃1≤MULT(U,SETSEQ(M)))⊃(∀M.M<LENGTH U⊃1=MULT(U,SETSEQ(M))))
(label pigeonlist) ■
```

8.19. file ALPIG: Application to Alists 1: Disjointness.

```
;first application: to alists. Lemma:inj implies disjoint
(wipe-out)
(get-proofs appal)
(proof inj_disj)

;a main lemma for the induction step

1.  (assume linj u|)
    (label injdsj0)

2.  (rw * (open inj))
    (label injdsjl)
    ;∀N M.N<LENGTH U∧M<LENGTH U∧NTH(U,N)=NTH(U,M)⊃N=M

3.  (assume |n<length u|)
    (label injdsj2)

4.  (assume|(un(λm.mkset(nth(u,m)),n))(xv)∧(mkset(nth(u,n)))(xv)|)
    (label injdsj3)

    ;need mksetfact

5.  (ue ((u.u)(n.n)) mksetfact (open lesseq) injdsj2)
    ;UN(λM.MKSET(NTH(U,M)),N)=(λX.(∃K.K<N∧NTH(U,K)=X))

6.  (rw injdsj3 (use * mode: exact) (open mkset) injdsj2)
    ;(∃K.K<N∧NTH(U,K)=XV)∧XV=NTH(U,N)
    (label injdsj4)

7.  (define kv|kv<n∧nth(u,kv)=xv| (use *))
    (label injdsj5)

8.  (derive |kv<length u∧nth(u,kv)=nth(u,n)|
          (* injdsj2 transitivity-of-order)
          (use injdsj4 mode: always direction: reverse>>

9.  (derive |kv=n| (injdsj2 * injdsj1))

10. (rw injdsj5 (use * mode: exact)  irreflexivity-of-order)
    ;FALSE
    ;deps: (INJDSJ0 INJDSJ3 INJDSJ2)

11. (ci injdsj3)
    ;¬((UN(λM.MKSET(NTH(U,M)),N))(XV)∧(MKSET(NTH(U,N)))(XV))

12. (ci (injdsj0 injdsj2))
    ;INJ(U)∧N<LENGTH U⊃¬((UN(λM.MKSET(NTH(U,M)),N))(XV)∧(MKSET(NTH(U,N)))(XV))
    (label injdsj-lemma)

    ;the theorem follows

13. (ue (a |λn.inj(u)∧n≤length(u)⊃disjoint(λm.mkset nth(u,m),n) |)
      proof-by-induction
      (open disjoint disj_pair intersection emptyp)
      (use less-lesseqsucc mode: always direction: reverse>
      (use injdsj-lemma mode: always)(part 1#2#1#1 (open lesseq)))
```

```
;∀N.INJ(U)∧N≤LENGTH U⊃DISJOINT(λM.MKSET(NTH(U,M)),N)

14. (ue (n llength u|) * (open lesseq))
    ;INJ(U)⊃DISJOINT(λM.MKSET(NTH(U,M)),LENGTH U)
(label inj_disj) ∎
```

8.20: Application to Alists 2: the Multiplicity is Positive.

```
;the sets in the sequence have positive multiplicity
(proofpermutp_injectp_lemma)

(assume lmklset u=mklset v|)
(label pil1)

2.  (assume I n<length u|)
    (label pil2)

    :use nthmember

3.  (trw |nth(u,n)∈ mklset u| (open epsilon mklset)
         nthmember pil2)
    ;NTH(U,N)∈MKLSET(U)
    ;deps: (PIL2)

    ;now use line pill

4.  (rw * (use pill mode: exact))
    ;NTH(U,N)∈MKLSET(V)
    ;deps: (PIL1 PIL2)

    ;Finally using MKLSET-FACT, we prove the existence of a kv such that
    ;nth(v,kv)=nth(u,n)

    ;labels: MKLSET-FACT
    ;∀U.MKLSET(U)=(λX.(∃K.K<LENGTH U∧NTH(U,K)=X))

5.  (rw * (use mklset-fact mode: exact> (open epsilon mkset))
    ;∃K.K<LENGTH V∧NTH(V,K)=NTH(U,N)
    ;deps: (PIL1 PIL2)

6.  (define kv|kv<length(v)∧nth(v,kv)=nth(u,n)| * )
    (label pil3)
    ;deps: (PIL1 PIL2)

7.  (trw |member(nth(v,kv),v)| nthmember pil3)
    ;MEMBER(NTH(V,KV),V)
    (label pil4)

    ;Therefore the set mkset(nth(u,n)) has positive multiplicity in v.

    ;labels : MEMBER_MULT
    ;∀U Y A.MEMBER(Y,U)∧A(Y)⊃1≤MULT(U,A)

8.  (ue ((u.v)(y.|nth(v,kv)|)(a.|mkset nth(u,n)|)) member-mult
        (part 1(open mkset)) pil2 pil4 (use pil3 mode: always))
    ;1≤MULT(V,MKSET(NTH(U,N)))
    ;deps: (PIL1 PIL2)

9.  (ci (pill pil2))
    ;MKLSET(U)=MKLSET(V)∧N<LENGTH U⊃1≤MULT(V,MKSET(NTH(U,N)))

    ;cosmetics
```

10. (derive |∀u v.mklset u=mklset v⊃(∀m.m<length u⊃1≤mult(v,mkset nth(u,m)))| *)
(label permutp_injectp_lemma) ∎

8.21. Application to Alists 3: Multiplicities in Dom and Range.

;lemma mult-mult
(proof mult_mult)

1. (assume |mklset u = mklset v|)
(label mm1)

2. (assume |∀m.m<length u ⊃ mult(v,mkset nth(u,m))=1|)
(label mm2)

3. (assume |i<length v|)
(label mm3)

4. (trw |nth(v,i) ∈ mklset v| (open epsilon mklset)
(use * nthmember mode: exact))
;NTH(V,I)∈MKLSET(V)

5. (rw * (use mm1 mode: exact direction: reverse>>
;NTH(V,I)∈MKLSET(U)

6. (rw * (use mklset_fact mode: exact) (open epsilon))
;∃K.K<LENGTH U∧NTH(U,K)=NTH(V,I)

7. (define mv |mv<length u ∧nth(u,mv)=nth(v,i)| *)
(label mm4)
;MV is unknown.
;the symbol MV is given the same declaration as M
;deps: (MM1 MM3)

8. (ue (m mv) mm2 (use * mode: always))
;MULT(V,MKSET(NTH(V,I)))=1
;deps: (MM1 MM2 MM3)

9. (ci mm3)
;I<LENGTH V⊃MULT(V,MKSET(NTH(V,I)))=1
;deps: (MM1 MM2)

10. (ci (mm1 mm2))
;MKLSET(U)=MKLSET(V)∧(∀M.M<LENGTH U⊃MULT(V,MKSET(NTH(U,M)))=1)⊃
;(I<LENGTH V⊃MULT(V,MKSET(NTH(V,I)))=1)
(label mult_mult) ∎

8.22. Application to Alists: a Permutation is an Injection.

;the main result for permutp: theorem permutp-injectp

(proof permutp_injectp)

1. (assume |permutp alist |)
(label permutp_injectp1)

2. (rw * (open permutp))
;FUNCTP(ALIST)∧MKLSET(DOM(ALIST))=MKLSET(RANGE(ALIST))

```
    (label permutp_injectp2)

3.  (rw * (open functp))
    ;UNIQUENESS(DOM(ALIST))∧MKLSET(DOM(ALIST))=MKLSET(RANGE(ALIST))
    (label permutp_injectp3)


    ;first step: disjointness of a suitable sequence of sets

    ;labels: UNIQUENESS-INJECTIVITY
    ;∀U.UNIQUENESS(U)≡INJ(U)

    ;labels: INJ_DISJ
    ;∀U.INJ(U)⊃DISJOINT(λM.MKSET(NTH(U,M)),LENGTH U)

4.  (derive |inj(dom(alist))| (* uniqueness_injectivity))
    ;deps: (PERMUTP_INJECTP1)

5.  (derive |disjoint(λm.mkset(nth(dom(alist),m)),length(dom(alist)))|
         (* inj_disj))
    (label permutp_injectp4)


    ;second step: multiplicity of the sets in the sequence is positive

    ;labels: PERMUTP_INJECTP_LEMMA
    ;∀U V.MKLSET(U)=MKLSET(V)⊃(∀M.M<LENGTH U⊃1≤MULT(V,MKSET(NTH(U,M))))

6.  (ue ((u.|dom alist|)(v.|range alist|)) permutp_injectp_lemma
       (permutp_injectp3 permutp_injectp4))
    ;∀M.M<LENGTH (DOM(ALIST))⊃1≤MULT(RANGE(ALIST),MKSET(NTH(DOM(ALIST),M)))
    (label permutp_injectp5)


    ;third step: application of the pigeon hole principle

    ;labels: PIGEONLIST
    ;∀SETSEQ U.DISJOINT(SETSEQ,LENGTH U)∧(∀K.K<LENGTH U⊃1≤MULT(U,SETSEQ(K)))⊃
    ;          (∀K.K<LENGTH U⊃1=MULT(U,SETSEQ(K))))

    ;need also
    ;labels: DOMRANGELENGTH
    ;∀ALIST.LENGTH (DOM(ALIST))=LENGTH (RANGE(ALIST))

7.  (ue ((setseq.|λm.mkset nth(dom alist,m)|)(u.|range alist|)) pigeonlist
       (use domrangelength mode: exact direction: reverse>
       permutp_injectp4 permutp_injectp5)
    ;∀K.K<LENGTH (DOM(ALIST))⊃1=MULT(RANGE(ALIST),MKSET(NTH(DOM(ALIST),K)))

    ;fourth step: injectivity

    ;labels: MULT_MULT
    ;∀U V.MKLSET(U)=MKLSET(V)∧(∀K.K<LENGTH U⊃MULT(V,MKSET(NTH(U,K)))=1)⊃
    ;     (∀I.I<LENGTH V⊃MULT(V,MKSET(NTH(V,I)))=1)

8.  (ue ((u.|dom(alist)|)(v.|range(alist)|)) mult_mult
       permutp_injectp3 * )
    ;∀I.I<LENGTH (RANGE(ALIST))⊃MULT(RANGE(ALIST),MKSET(NTH(RANGE(ALIST),I)))=1
    ;deps: (PERMUTP_INJECTP1)

    ;apply mult-inj

    ;labels: MULT_INJ
    ;∀V.(∀K.K<LENGTH V⊃MULT(V,MKSET(NTH(V,K)))=1)⊃INJ(V)

9.  (ue (v |range alist|) mult-inj * )
```

```
;INJ(RANGE(ALIST))
;deps:(PERMUTP_INJECTP1)

10. (derive |uniqueness(range alist)| (* uniqueness_injectivity))
    ;deps:(PERMUTP_INJECTP1)

11. (derive |injectp alist | (permutp_injectp2 *)(open injectp))
    ;deps: (PERMUTP_INJECTP1

12. (ci (permutp_injectp1))
    ;PERMUTP(ALIST)⊃INJECTP(ALIST)
    (label theorem_permutp_injectp)     ■

    (save-proofs alpig)
```

8.23. file LPIG: Application to Lists 1: Disjointness.

```
;Disjointness
(proof disjoint-number)

;lemma dn1

1.  (ue (a |λn.∀m.(un((λxv.mkset(xv)),n))(m)⊃m<n|)
       proof-by-induction
       (part 1(open mkset un emptyset union))
       (use normal mode: always)
       (use successor1 transitivity-of-order))
    ;∀N M.(UN(λXV.MKSET(XV),N))(M)⊃M<N

    ;lemma disjoint number

2.  (ue ((n.n)(m.n)) dnl irreflexivity-of-order)
    ;¬(UN(λXV.MKSET(XV),N))(N)

3.  (trw |(un(λyv.mkset(yv),n))(xv)∧(mkset(n))(xv)| * (part 2(open mkset)))
    ;¬((UN(λYV.MKSET(YV),N))(XV)∧(MKSET(N))(XV))

4.  (ue (a |λn.disjoint(λxv.mkset(xv),n)|) proof-by-induction
       (open disjoint disj_pair emptyp intersection)
       (use * mode: exact))
    ;∀N.DISJOINT(λXV.MKSET(XV),N)
    (label disjoint-number) ■
```

8.24. Application to Lists 3: Multiplicity in the Range.

```
;lemma into-mult

(proof into-mult)

1. (assume |into(u)|)
    label im1)

2.  (assume |∀k.k<length u⊃1=mult(u,mkset k)|)
    (label im2)

3.  (assume |i<length u|)
    (label im3)
```

```
4.  (rw iml (open into))
    ;∀N.N<LENGTH U⊃NATNUM(NTH(U,N))∧NTH(U,N)<LENGTH U
    ;deps: (IM1)

5.  (ue (k |nth(u,i)|) im2 (use im3 * mode: exact))
    ;1=MULT(U,MKSET(NTH(U,I)))
    ;deps: (IM1 IM2 IM3)

6.  (ci im3)
    ;I<LENGTH U⊃1=MULT(U,MKSET(NTH(U,I)))
    ;deps: (IM1 IM2)

7.  (ci (im1 im2))
    ;INTO(U)∧(∀K.K<LENGTH U⊃1=MULT(U,MKSET(K)))⊃
    ;(I<LENGTH U⊃1=MULT(U,MKSET(NTH(U,I))))
    (label into-mult) ■
```

8.25. Application to Lists: a Permutation is an Injection.

```
    ;the main result for perm
    ;a straightforward application of pigeon hole to onto lists

    (proof perm_inj)
    ;∀U.PERM(U)⊃INJ(U)

1.  (assume |perm u|)
    (label perm_inj1)

2.  (rw * (open perm onto))
    ;INTO(U)∧(∀N.N<LENGTH U⊃MEMBER(N,U))
    (label perm_inj2)

    ;labels: MEMBER_MULT
    ;∀U Y A.MEMBER(Y,U)∧A(Y)⊃1≤MULT(U,A)

3.  (ue ((u.u)(y.n)(a.|mkset n|)) member-mult
        (part 1(open mkset)))
    ;MEMBER(N,U)⊃1≤MULT(U,MKSET(N))

4.  (derive |∀n.n<length u⊃1≤mult(u,mkset n)| (perm_inj2 *))
    (label onto_mult)(label perm_inj3)
    ;deps: (PERM_INJ1)

5.  (ue((setseq.|λxv.mkset(xv)|)(u.u)) pigeonlist disjoint-number perm_inj3)
    ;∀K.K<LENGTH U⊃1=MULT(U,MKSET(K))
    (label perm_inj4)
    ;deps: (PERM_INJ1)

    ;labels: INTO-MULT
    ;∀U.INTO(U)∧(∀K.K<LENGTH U⊃1=MULT(U,MKSET(K)))⊃
    ,           (∀I.I<LENGTH U⊃1=MULT(U,MKSET(NTH(U,I))))

6.  (derive |∀i.i<length u⊃1=mult(u,mkset(nth(u,i)))| (into-mult perm_inj2 *))
    ;deps: (PERM_INJ1)

    ;labels: MULT_INJ
    ;∀V.(∀K.K<LENGTH V⊃MULT(V,MKSET(NTH(V,K)))=1)⊃INJ(V)

7.  (ue (v u) mult_inj * )
    ;INJ(U)
    ;deps: (PERM_INJ1)
```

```
8.  (ci perm_inj1)
    ;PERM(U)⊃INJ(U)
    (label  perm_injectvity) ■

    (save-proof lpig)
```

8.26. Operations on Functions Represented by Association Lists.

```
    ;the approach using association lists
    (wipe-out)
    (get-proofs appal)

    (proof assoc)
1.  (decl (compalist) (infixname: |∞|) (type: |ground•ground→ground|)
         (syntype: constant) (bindingpower: 930))
2.  (defax  compalist
          |∀alist1 alist2 xa y.nil ∞ alist2=nil∧
                               ((xa.y).alist1) ∞ alist2=
                                (xa.appalist(y,alist2)).(alist1 ∞ alist2)|)
    (label  compalistdef)

3.  (decl invalist (type: |ground→ground|))
4.  (defax  invalist
          |∀alist xa y.invalist nil=nil∧
                       invalist((xa.y).alist)=(y.xa).invalist alist|)
    (label  invalistdef)

5.  (decl idalistp (type: |ground→truthval|))
6.  (defax  idalistp
          |∀alist x a  y.idalistp(nil)∧
                       (idalistp((xa.y).alist)≡xa=y∧idalistp alist)|)
    (label  idalistpdef)
```

8.26.1. file ASSOC: Functions Represented by Association Lists.

```
    (proof alistprop)

    ;prove sorts

    ;compalist sort

1.  (ue (chi |λalist.alistp(alist ∞ alist1)|)   alistinduction
        (part 1(open compalist)(use appalistsort mode: exact>>>
    ;∀ALIST.ALISTP ALIST ∞ ALIST1

    (label simpinfo) (label compalistsort)     ■

    ;invalistsort

2.  (ue (chi |λalist.allp(λx.atom x,range alist)⊃alistp invalist(alist)|)
        alistinduction (open range member invalist)
        (use allpfact ue: ((phi.|λx.atom x|)(x.y)(u.|range alist|)) mode: always) )
    ;∀ALIST.ALLP(λX.ATOM X,RANGE(ALIST))⊃ALISTP INVALIST(ALIST)
    (label invalistsort) ■

    ;prove facts about composition of functions

    ;three (of five) lemmata
```

```
;lemma 1

3.  (ue (chi |λalist.member(x,dom(alist))⊃
            appalist(x,alist m alist1)=appalist(appalist(x,alist),alist1)|)
      alistinduction
      (part 1(use appalistdef mode: always)
             (open dom member compalist assoc))
             (use normal mode: always))
    ;∀ALIST.MEMBER(X,DOM(ALIST))⊃
    ;       APPALIST(X,ALIST m ALIST1)=APPALIST(APPALIST(ALIST,X),ALIST1)
    (label alist_lemma1) (label app_compalist)     ■

    ;lemma 2

4.  (ue (chi |λalist.dom(alist m alist1)=dom(alist)|)
      alistinduction
      (open compalist dom))
    ;∀ALIST.DOM(ALIST m ALIST1)=DOM(ALIST)
    (label alist_lemma2) (label dom_compalist)     ■

    ;compalist lemma

5.  (ue (chi |λalist.¬member(za,range alist)⊃alist m ((za.z).alist1)=alist m alist1|)
      alistinduction
      (open member range compalist appalist assoc) (use demorgan mode: always))
    ;∀ALIST.¬MEMBER(ZA,RANGE(ALIST))⊃ALIST m ((ZA.Z).ALIST1)=ALIST m ALIST1
    (label compalist-lemma) ■

    ;samemap right

6.  (ue (chi |λalist.samemap(alist1,alist2)⊃alist m alist1=alist m alist2|)
      alistinduction
      (part 1(use samemap_def1 mode: exact))
      (part 1(open compalist samemap)))
    ;∀ALIST.SAMEMAP(ALIST1,ALIST2)⊃ALIST m ALIST1=ALIST m ALIST2
    (label samamap,right) ■

    ;prove a fact about the identity function

    ;idalistp_main

7.  (ue (chi |λalist.idalistp(alist)∧member(y,dom alist)⊃appalist(y,alist)=y|)
      alistinduction
      (open idalistp appalist assoc member dom) (use normal mode: always))
    ;∀ALIST.IDALISTP(ALIST)∧MEMBER(Y,DOM(ALIST))⊃CDR ASSOC(Y,ALIST)=Y
    (label idalistp-main) ■

    ;prove facts about inversion of functions

    ;dom invalist

8.  (ue (chi |λalist.allp(λx.atom x,range alist)⊃dom invalist(alist)=range alist|)
     alistinduction (open dom range invalist) (use invalistsort)
     (use allpfact ue: ((phi.|λx.atom x|)(x.y)(u.|range alist|)) mode: always) )
    ;∀ALIST.ALLP(λX.ATOM X,RANGE(ALIST))⊃DOM(INVALIST(ALIST))=RANGE(ALIST)
    (label  dom_invalist)  ■

    ;range invalist

9.  (ue (chi |λalist.allp(λx.atom x,range alist)⊃range invalist(alist)=dom alist|)
      alistinduction (open dom range invalist) (use invalistsort)
      (use allpfact ue: ((phi.|λx.atom x|)(x.y)(u.|range alist|)) mode: always) )
    ;∀ALIST.ALLP(λX.ATOM X,RANGE(ALIST))⊃RANGE(INVALIST(ALIST))=DOM(ALIST)
    (label range-invalist) ■
```

8.26.2. Lemma Nonempty Range.

```
(proof nonempty-range)
;lemma 3

5. (ue (chi |λalist.member(x,dom alist)⊃somep(λy.appalist(x,alist)=y,range alist)|)
       alistinduction
       (part 1 (open dom somep range member appalist assoc))
       (use normal mode: always))
   ;∀ALIST.MEMBER(X,DOM(ALIST))⊃SOMEP(λY.APPALIST(X,ALIST)=Y,RANGE(ALIST))

6. (rw * (use somepfact mode: exact))
   ;∀ALIST.MEMBER(X,DOM(ALIST))⊃
   ;      (∃X1.MEMBER(X1,RANGE(ALIST))∧APPALIST(X,ALIST)=X1)
   (label nonempty-range) ∎
```

8.26.3. Lemma Nonempty Domain.

This lemma says that if z belongs to range(alist), then there is an x in dom(alist) such that appalist (x, alist) = z. As noticed above, this requires the fact that alist represents a function, i.e. that dom(alist) has the *uniqueness* property, for if some (x z1) occurs in alist before (x z), with z1≠z, then appalist(x,alist) will give z1 as value.

```
;lemma 4
(proof nonempty-domain)

1. (assume |uniqueness dom(alist)∧member(z,range alist)⊃
           (∃x.member(x,dom alist)∧appalist(x,alist)=z)|)
   (label lem41)

2. (assume |uniqueness dom((xa.y).alist)|)
   (label lem42)

3. (rw * (open uniqueness dom))
   ;¬MEMBER(XA,DOM(ALIST))∧UNIQUENESS(DOM(ALIST))
   (label lem43)
   ;deps: (LEM42)

4. (assume|member(z,range((xa.y).alist))|)
   (label lem44)

5. (rw * (open range member))
   ;Z=Y∨MEMBER(Z,RANGE(ALIST))
   (label lem45)
   ;deps: (LEM44)
```

We use the last line for a proof by cases. The first case follows by expanding the definitions.

```
6. (assume |z=y|)

7. (trw |∃x1.member(x1,dom((xa.y).alist))∧appalist(x1,(xa.y).alist)=z|
        (open dom member appalist assoc) (use * mode: exact))
   ;∃X1.MEMBER(X1,DOM((XA.Y).ALIST))∧APPALIST(X1,(XA.Y).ALIST)=Z
   (label lem46)
```

We use the assumption of the second case and the induction hypothesis (line 1) to get an element x_z in the inverse image of z (line 9); then we use the assumption of *uniqueness* (lines 2 and 3) to show that $x_z \neq$ xa (line 10) and

$$\text{appalist}\ (x_z\ ,(\text{xa.y}).\text{alist}) = \text{appalist}(x_z,\text{alist}) = z$$

(line 11).

```
8.  (assume |member(z,range(alist))|)
    (label lem47)

9.  (define xxv |member(xxv,dom alist)∧appalist(xxv alist)=z|
           (lem41 lem43 lem47))
    (label lem48)
    ;deps: (LEM41 LEM42 LEM47)

10. (derive |xxv≠xa| (lem43 lem48))
  , ;deps: (LEM41 LEM42 LEM47)

11. (trw |appalist(xxv,(xa.y).alist)=z|  (open appalist assoc)
        (use * mode: exact)(use lem48 mode: always direction: reverse>>
    ;APPALIST(XXV,(XA.Y).ALIST)=Z
    ;deps: (LEM41 LEN42 LEM47)

12. (derive
      |∃x1.member(x1,dom((xa.y).alist))∧appalist(x1,(xa.y).alist)=z|
      (lem48 *) (open dom) (use memberdef mode: always))
    (label lem49)
    ;deps: (LEM41 LEH42 LEM47)

13. (cases lem45 lem46 lem49)
    ;∃X1.MEMBER(X1,DOM((XA.Y).ALIST))∧APPALIST(X1,(XA.Y).ALIST)=Z
    ;deps: (LEM41 LEM42 LEM44)

14. (ci (lem42 lem44))
    ;UNIQUENESS(DOM((XA.Y).ALIST))∧MEMBER(Z,RANGE((XA.Y).ALIST))⊃
    ;(∃X1.MEMBER(X1,DOM((XA.Y).ALIST))∧APPALIST(X1,(XA.Y).ALIST)=Z)
    ;deps: (LEM41)

15. (ci lem41)

16. (ue (chi |λalist.uniqueness dom(alist)∧member(z,range alist)⊃
                  (∃x.member(x,dom alist)∧appalist(x,alist)=z)|)
      alistinduction
      (part 1#1 (open range member)) (use * mode: exact))
    ;∀ALIST.UNIQUENESS(DOM(ALIST))∧MEMBER(Z,RANGE(ALIST))⊃
    ,      (∃X.MEMBER(X,DOM(ALIST))∧APPALIST(X,ALIST)=Z)
    (label nonempty-domain) ■
```

8.26.4. Lemma Range Compose, Part 1.

```
;theorem 1 (i); lemma range compose, part 1
    (proof range-compose)

1.  (assume|permutp(alist)|)
    (label rc1)

2.  (rw * (open permutp functp))
    ;UNIQUENESS(DOM(ALIST))∧MKLSET(DOM(ALIST))=MKLSET(RANGE(ALIST))
    (label rc2)
```

```
3.  (assume lmklset dom(alist)=mklset dom(alist1)|)
    (label rc3)

4.  (assume |member(z,range(alist o alist1))|)
    (label rc4)

    ;apply lemma 4 and lemma 2

    (ue ((alist.|alist o alist1|)(z.z)) nonempty-domain
         (use dom_compalist rc2 rc4 mode: exact) )
    ;deps: (RC1 RC4)
    ;∃X.MEMBER(X,DOM(ALIST))∧APPALIST(X,ALIST o ALIST1)=Z

6.  (define xxvv |member(xxvv,dom alist)∧appalist(xxvv,alist m alist1)=z| * )
    (label rc5)
    ;deps: (RC1 RC4)

    ;apply lemma 1

7.  (rw * (use app_compalist mode: always))
    ;MEMBER(XXVV,DOM(ALIST))∧APPALIST(APPALIST(XXVV,ALIST),ALIST1)=Z
    (label rc6)
    ;deps: (RC1 RC4)

    ;apply lemma 3

8   (define yyvv |member(yyvv,range alist)∧appalist(xxvv,alist)=yyvv|
           (nonempty-range rc6))
    (label rc7)
    ;deps: (RC1 RC4)

9.  (trw lyyvv ∈ mklset range(alist)| (open mklset epsilon) rc7)
    ;YYVV∈MKLSET(RANGE(ALIST))
    ;deps: (RC1 RC4)

10  (rw * (use rc2 mode: exact direction: reverse>
        (use rc3 mode: exact))
    ;YYVV∈MKLSET(DOM(ALIST1))
    ;deps: (RC1 RC3 RC4)

11. (rw * (open epsilon mklset))
    ;MEMBER(YYVV,DOM(ALIST1))
    ;deps: (RC1 RC3 RC4)

    ;apply again lemma 3, this time to alistl

12  (define zzvv |member(zzvv,range alist1)∧appalist(yyvv,alist1)=zzvv|
           (nonempty-range *))
    (label rc8)
    ;deps: (RC1 RC3 RC4)

13. (rw rc6 rc7)
    ;MEMBER(XXVV,DOM(ALIST))∧APPALIST(YYVV,ALIST1)=Z
    ;deps: (RC1 RC4)

14. (trw |zzvv=z| * (use rc8 mode: always direction: reverse>>
    ; ZZVV=Z
    ;deps: (RC1 RC3 RC4)

15. (trw |member(z,range alist1)| rc8 (use * mode: exact direction: reverse>>
    ;MEMBER(Z,RANGE(ALIST1))
    ;deps: (RC1 RC3 RC4)

16. (ci rc4)
    ;MEMBER(Z,RANGE(ALIST o ALIST1))⊃MEMBER(Z,RANGE(ALIST1))
```

```
    ;deps: (RC1 RC3)

17. (trw lmklset range(alist ∘ alist1)⊂mklset range(alist1)| *
        (open mklset inclusion))
    ;MKLSET(RANGE(ALIST ∘ ALIST1))⊂MKLSET(RANGE(ALIST1))
    ;deps: (RC1 RC3)

18. (ci (rc1 rc3))
    ;PERMUTP(ALIST)∧MKLSET(DOM(ALIST))=MKLSET(DOM(ALIST1))⊃
    ;MKLSET(RANGE(ALIST ∘ ALIST1))⊂MKLSET(RANGE(ALIST1))
```

8.26.5. Lemma Range Compose, Part 2.

```
;theorem 1 (i); lemma range compose, part 2
    (proof range_compose2)

1.  (assume|permutp(alist)|)
    (label rc21)

2.  (rw * (open permutp functp))
    ;UNIQUENESS(DOM(ALIST))∧MKLSET(DOM(ALIST))=MKLSET(RANGE(ALIST))
    (label rc22)

3.  (assume|permutp(alist1)|)
    (label rc23)

4.  (rw * (open permutp functp))
    ;UNIQUENESS(DOM(ALIST1))∧MKLSET(DOM(ALIST1))=MKLSET(RANGE(ALIST1))
    (label rc24)

5.  (assume |mklset dom(alist)=mklset dom(alist1)|)
    (label rc25)

6.  (assume |member(z,range alist1)|)
    (label rc26)

    ;apply lemma 4

7.  (define yvl |member(yv1,dom alist1)∧appalist(yv1,alist1)=z|
            (nonempty-domain rc24 rc26))
    (label rc27)
    ;deps: (RC23 RC26)

8.  (trw |yv1 ∈ mklset dom(alist1)| * (open epsilon mklset))
    ;YV1∈MKLSET(DOM(ALIST1))
    ;deps: (RC23 RC26)

9.  (rw * (use rc25 mode: exact direction: reverse)
        (use rc22 mode: exact>>
    ;YV1∈MKLSET(RANGE(ALIST))
    ;deps: (RC21 RC23 RC25 RC26)

10  (rw * (open epsilon mklset))
    ;MEMBER(XV1,RANGE(ALIST))
    (label rc28)
    ;deps: (RC21 RC23 RC25 RC26)

    ;apply again lemma 4, this time to alist

11. (define xv1 |member(xv1,dom alist)∧appalist(xv1,alist)=yv1|
            (nonempty-domain rc22 rc28))
    (label rc29)
```

```
;deps: (RC21 RC23 RC25 RC26)

;apply lemma 2 and rewrite

12. (trw |member(xv1,dom(alist ∞ alist1))| * (use dom_compalist))
    ;MEMBER(XV1,DOM(ALIST ∞ ALIST1))
    (label rc30)
    ;deps: (RC21 RC23 RC25 RC26)

13. (trw |appalist(xv1,alist m alist1) I rc29 rc30
         (use app_compalist rc29 rc27 mode: always))
    ;APPALIST(XV1,ALIST ∞ ALIST1)=Z
    (label rc31)
    ;deps: (RC21 RC23 RC25 RC26)

;apply lemma 3

14. (ue ((alist.|alist m alist1|)(x.xv1)) nonempty-range
        (use dom_compalist rc22 rc30 mode: always))
    ;∃Y.MEMBER(Y,RANGE(ALIST m ALIST1))∧APPALIST(XV1,ALIST ∞ ALIST1)=Y
    ;deps: (RC21 RC23 RC25 RC26)

15. (define zvl
           |member(zv1,range(alist m alist1))∧appalist(xv1,alist m alist1)=zv1| * )
    (label rc32)
    ;deps: (RC21 RC23 RC25 RC26)

16. (trw |zv1=z| rc31 (use * mode: always direction: reverse>>
    ;ZV1=Z
    ;deps: (RC21 RC23 RC25 RC26)

17. (trw |member(z,range(alist ∞ alist1))| rc32
         (use * mode:  exact direction: reverse>>
    ;MEMBER(Z,RANGE(ALIST m ALIST1))
    ;deps: (RC21 RC23 RC25 RC26)

18. (ci rc26)
    ;MEMBER(Z,RANGE(ALIST1))⊃MEMBER(Z,RANGE(ALIST ∞ ALIST1))
    ;deps: (RC21 RC23 RC25)

19. (trw |mklset range(alist1)⊂mklset range(alist ∞ alist1)| *
         (open inclusion mklset) )
    ;MKLSET(RANGE(ALIST1))⊂MKLSET(RANGE(ALIST m ALIST1))
    ;deps: (RC21 RC23 RC25)

20. (ci (rc21 rc23 rc25))
    ;PERMUTP(ALIST)∧PERMUTP(ALIST1)∧MKLSET(DOM(ALIST))=MKLSET(DOM(ALIST1))⊃
    ;MKLSET(RANGE(ALIST1))⊂MKLSET(RANGE(ALIST ∞ ALIST1))
```

8.26.6. Conclusion of Theorem 1.

```
(proof permutp_compalist)

1.  (assume|permutp(alist)|)
    (label permut_comp1)

2.  (assume|permutp(alist1)|)
    (label permut_comp2)

3.  (assume|mklset(dom(alist))=mklset(dom(alist1))|)
    (label permut_comp3)
```

```
4.  (derive |mklset(range(alist ∘ alist1))⊂mklset(range(alist1))∧
            mklset(range(alist1))⊂mklset(range(alist ∘ alist1))|
           (permut-compl permut_comp2 permut_comp3 range-compose))
    ;deps: (PERMUT_COMP1 PERMUT_COMP2 PERMUT_COMP3)

5.  (derive |mklset(range(alist ∘ alist1))=mklset(range(alist1))|
           (* double-inclusion))
    ;deps: (PERMUT_COMP1 PERMUT_COMP2 PERMUT_COMP3)
    (label permut_comp4)

6.  (rw permut-compl (open permutp functp))
    ;UNIQUENESS(DOM(ALIST))∧MKLSET(DOM(ALIST))=MKLSET(RANGE(ALIST))
    (label permut_comp5)

7.  (rw permut_comp2 (open permutp))
    ;FUNCTP(ALIST1)∧MKLSET(DOM(ALIST1))=MKLSET(RANGE(ALIST1))

8.  (trw |uniqueness(dom(alist ∘ alist1))∧
          mklset dom(alist ∘ alist1)=mklset range(alist m alist1)|
         (use dom_compalist permut_comp4 mode: exact) permut_comp5
         (use * permut_comp3 mode: always direction: reverse>>
    ;UNIQUENESS(DOM(ALIST ∘ ALIST1))∧
    ;MKLSET(DOM(ALIST ∘ ALIST1))=MKLSET(RANGE(ALIST m ALIST1))
    ;deps: (PERMUT_COMP1 PERMUT_COMP2 PERMUT_COMP3)

9.  (trw |permutp(alist ∘ alist1)|* (open permutp functp))
    ;PERMUTP(ALIST ∘ ALIST1)
    ;deps: (PERMUT_COMP1 PERMUT_COMP2 PERMUT_COMP3)

10. (ci (permut-compl permut_comp2 permut_comp3))
    ;PERMUTP(ALIST)∧PERMUTP(ALIST1)∧MKLSET(DOM(ALIST))=MKLSET(DOM(ALIST1))⊃
    ;PERMUTP(ALIST m ALIST1)
    (label  permutp_compalist) ■
```

8.26.7. Associativity of Composition.

```
    ;theorem 1 (ii)
    (proof compalist_associativity)

1.  (trw |mklset(range((xa.y).alist))⊂mklset(dom alist1)⊃
          member(y,dom alist1)∧mklset range(alist)⊂mklset dom(alist1)|
         (open mklset inclusion range member)(use normal mode: always))
    ;MKLSET(RANGE((XA.Y).ALIST))⊂MKLSET(DOM(ALIST1))⊃
    ;MEMBER(Y,DOM(ALIST1))∧MKLSET(RANGE(ALIST))⊂MKLSET(DOM(ALIST1))

2.  (trw |member(y,dom alist1)∧mklset range(alist)⊂mklset dom(alist1)⊃
          mklset(range((xa.y).alist))⊂mklset(dom alistl) |(der)
         (open mklset inclusion range member)(use normal mode: always))
    ;MEMBER(Y,DOM(ALIST1))∧MKLSET(RANGE(ALIST))⊂MKLSET(DOM(ALIST1))⊃
    ;MKLSET(RANGE((XA.Y).ALIST))⊂MKLSET(DOM(ALIST1))

3.  (derive |mklset(range((xa.y).alist))⊂mklset(dom alist1)≡
            member(y,dom alist1)∧mklset range(alist)⊂mklset dom(alist1)| (* -2))
    (label helpinduction)

4.  (ue (chi |λalist.mklset(range alist)⊂mklset(dom alist1)⊃
                    alist ∘ (alistl m alist2)=(alist m alistl) ∘ alist2|)
       alistinduction
       (part 1(open compalist)(use app_compalist * mode: always)))
    ;∀ALIST.MKLSET(RANGE(ALIST))⊂MKLSET(DOM(ALIST1))⊃
    ,       ALIST ∘ (ALIST1 ∘ ALIST2)=(ALIST m ALIST1) ∘ ALIST2
    (label   compalist_associativity)  ■
```

8.26.8. Samemap Left.

```
    (proof samemap_left)

1.  (assume|samemap(alist1,alist2)|)
    (label sml1)

2.  (rw * (open samemap))
    ;MKLSET(DOM(ALIST1))=MKLSET(DOM(ALIST2))∧
    ;(∀Y.Y∈MKLSET(DOM(ALIST1))⊃APPALIST(Y,ALIST1)=APPALIST(Y,ALIST2))
    (label sml2)

3.  (assume |y∈mklset dom(alist1)|)
    (label sml3)

4.  (derive |appalist(y,alist1)=appalist(y,alist2)| (sml2 sml3))
    (label sml4)

5.  (rw sml3 (use sml2 mode: exact))
    ;Y∈MKLSET(DOM(ALIST2))
    (label sml5)

6.  (rw sml3 (open epsilon mklset))
    ;MEMBER(Y,DOM(ALIST1))

7.  (rw sml5 (open epsilon mklset))
    ;MEMBER(Y,DOM(ALIST2))

8.  (trw |appalist(y,alist1 ∞ alist)=appalist(y,alist2 ∞ alist)|
         (use app-compalist -2 mode: exact)
         (use app-compalist * mode: exact)
         (use sml4 mode: exact))
    ;APPALIST(Y,ALIST1 m ALIST)=APPALIST(Y,ALIST2 ∞ ALIST)
    ;deps: (SML1 SML3)

9.  (ci sml3)
    ;Y∈MKLSET(DOM(ALIST1))⊃APPALIST(Y,ALIST1 m ALIST)=APPALIST(Y,ALIST2 ∞ ALIST)

10. (trw |mklset(dom(alist1 ∞ alist))=mklset(dom(alist2 ∞ alist))|
        dom_compalist (use sml2 mode: exact))
    ;MKLSET(DOM(ALIST1 m ALIST))=MKLSET(DOM(ALIST2 m ALIST))

11. (trw |samemap(alist1 ∞ alist,alist2 ∞ alist)| (open samemap)
        (dom_compalist * -2))
    ;SAMEMAP(ALIST1 ∞ ALIST,ALIST2 ∞ ALIST)
    ;deps: (SML1)

12. (ci sml1)
    ;SAMEMAP(ALIST1,ALIST2)⊃SAMEMAP(ALIST1 m ALIST,ALIST2 ∞ ALIST)
    (label samemap-left) ∎
```

8.26.9. Theorem 2, on Identity Alist.

```
    ;theorem 2 (i) (permutp idalistp)
    (proof idalistprop)

1.  (ue (chi |λalist.idalistp(alist)⊃dom alist=range alist|) alistinduction
        (open idalistp dom range))
    ;∀ALIST.IDALISTP(ALIST)⊃DOM(ALIST)=RANGE(ALIST)

2.  (trw |∀alist.functp(alist)∧idalistp(alist)⊃permutp(alist)|
```

```
        (open tunctp permutp)(use * mode: always))
    ;∀ALIST.FUNCTP(ALIST)∧IDALISTP(ALIST)⊃PERMUTP(ALIST)
    (label  idalistp_permutp)  ■

    ;theorem 2 (ii) (idalistp right)

3.  (assume|idalistp(alist1)|)

4.  (ue (chi |λalist.mklset(range(alist))⊂mklset(dom(alist1))⊃
                  (alist ∞ alist1=alist)|)
       alistinduction
       (part l(open compalist))
     (use helpinduction idalistp-main * mode: always))
    ;∀ALIST.MKLSET(RANGE(ALIST))⊂MKLSET(DOM(ALIST1))⊃ALIST m ALIST1=ALIST
    ;deps: (4)

5.  (ci -2)
    ;IDALISTP(ALIST1)⊃
    ;(∀ALIST.MKLSET(RANGE(ALIST))⊂MKLSET(DOM(ALIST1))⊃ALIST ∞ ALIST1=ALIST)
    (label  idalistp,right) ■

    ;theorem 2 (iii> (idalistp left)
    (proof  idalistp-left)

1.  (assume lidalistp alistid|)
    (label idal_l1)
    ;ALISTID is unknown.
    ;the symbol ALISTID is given the same declaration as ALIST

2.  (assume lmklset dom(alistid)=mklset dom(alist)|)
    (label idal_l2)

3.  (assume |y∈mklset(dom(alistid ∞ alist))|)
    (label idal_l3)

4.  (rw * (use dom_compalist mode: exact)(open epsilon mklset))
    (label idal_l4)
    ;MEMBER(Y,DOM(ALISTID))
    ;deps: (idal_l3)

5.  (trw |appalist(y,alistid ∞ alist)| (use app,compalist * mode: exact))
    ;APPALIST(Y,ALISTID m ALIST)=APPALIST(APPALIST(Y,ALISTID),ALIST)
    (label idal_l5)

    ;labels: IDALISTP_MAIN
    ;∀ALIST Y.IDALISTP(ALIST)∧MEMBER(Y,DOM(ALIST))⊃APPALIST(Y,ALIST)=Y

6.  (derive |appalist(y,alistid)=y| (idalistp-main idal_l1 idal_l4))
    ;deps: (idal_l1 idal_l3)

7.  (rw idal_l5 * )
    ;APPALIST(Y,ALISTID ∞ ALIST)=APPALIST(Y,ALIST)
    ;deps: (idal_l1 idal_l3)

8.  (ci idal_l3)
    ;Y∈MKLSET(DOM(ALISTID ∞ ALIST))⊃APPALIST(Y,ALISTID ∞ ALIST)=APPALIST(Y,ALIST)
    (label idal_l6)
    ;deps: (idal_l1)

9.  (trw |mklset(dom(alistid ∞ alist))=mklset dom(alist)|
        (use dom_compalist idal_l2 mode: exact))
    ;MKLSET(DOM(ALISTID ∞ ALIST))=MKLSET(DOM(ALIST))
    ;deps: (idal_l2)

    ;labels: SAMEMAPDEF
    ;∀ALIST ALIST1.SAMEMAP(ALIST,ALIST1)≡
```

```
;              MKLSET(DOM(ALIST))=MKLSET(DOM(ALIST1))∧
;              (∀Y.Y∈MKLSET(DOM(ALIST))⊃
;                 APPALIST(Y,ALIST)=APPALIST(Y,ALIST1))

10. (tra |samemap(alistid ∘ alist,alist)| (open samemap) (idal_16 *))
    ;SAMEMAP(ALISTID ∘ ALIST,ALIST)
    ;deps: (idal_11 idal_12)

11. (ci (idal_11 idal_12))
    ;IDALISTP(ALISTID)∧MKLSET(DOM(ALISTID))=MKLSET(DOM(ALIST))⊃
    ;SAMEMAP(ALISTID ∘ ALIST,ALIST)
    (label idalistp-left) ■
```

8.26.10. Lemma Atomrange.

```
;a lemma: the range of a permutation contains only atoms
(proof atomrange)

1.  (assume|mklset(dom(alist))=mklset(range(alist))|)
    (label ar1)

2.  (ue (chi |λalist.allp(λx.atom(x),dom alist)|)
       alistinduction
       (open allp dom))
    ;∀ALIST.ALLP(λX.ATOM X,DOM(ALIST))
    (label ar2)

3.  (ue ((phi1.|λx.atom(x)|)(x.x)(u.|dom alist |>> allp_elimination * )
    ;MEMBER(X,DOM(ALIST))⊃ATOM X

4.  (tra |mklset dom(alist)⊂(λx.atom x)|* (open inclusion mklset) )
    ;MKLSET(DOM(ALIST))⊂(λX.ATOM X)

5.  (ra * (use ar1 mode: exact))
    ;MKLSET(RANGE(ALIST))⊂(λX.ATOM X)

6.  (rn * (open inclusion mklset))
    ;∀XV.MEMBER(XV,RANGE(ALIST))⊃ATOM XV

7.  (ue ((phi1.|λx.atom x|)(u.|range alist|))
       allp_introduction * )
    ;ALLP(λX.ATOM X,RANGE(ALIST))

8.  (ci ar1)
    ;MKLSET(DOM(ALIST))=MKLSET(RANGE(ALIST))⊃ALLP(λX.ATOM X,RANGE(ALIST))
    (label atomrange) ■
```

8.26.11. Theorem 3, on Inversion of Alists.

```
;theorem 3 (i)
(proof permutp-invalist)

;we borrow this result from the proof permutp_injectp

;labels: PERMUTP_INJECTP
;∀ALIST.PERMUTP ALIST⊃INJECTP ALIST

(proof permutp-invalist)
```

```
1.  (assume lpermutp alist|)
    (label piv1)

2.  (derive |injectp alist|(permutp_injectp piv1))
    ;deps: (PIV1)

3.  (rw * (open injectp))
    ;FUNCTP(ALIST)∧UNIQUENESS(RANGE(ALIST))
    (label piv2)

4.  (rw pivl (open permutp))
    ;FUNCTP(ALIST)∧MKLSET(DOM(ALIST))=MKLSET(RANGE(ALIST))
    (label piv3)

5.  (derive |allp(λx.atom x,range alist)| (atomrange *))
    (label piv4)

6.  (derive ldom invalist(alist)=range alist|(dom_invalist *))
    (label piv5)

7.  (derive |range invalist(alist)=dom alist| (range-invalist piv4))
    (label piv6)

8.  (trw Iuniqueness dom(invalist(alist))| piv2 (use piv5))
    ;UNIQUENESS(DOM(INVALIST(ALIST)))
    (label piv7)

9.  (trw lmklset dom(invalist(alist))=mklset range(invalist(alist))|
 piv3 (use piv5 piv6))
    ;MKLSET(DOM(INVALIST(ALIST)))=MKLSET(RANGE(INVALIST(ALIST)))
    (label piv8)

10. (trw lpermutp invalist(alist)| piv7 piv8
 (open permutp functp) (use invalistsort piv4 mode: exact))
    ;PERMUTP(INVALIST(ALIST))
    ;deps: (PIV1)

11. (ci piv1)
    ;PERMUTP(ALIST)⊃PERMUTP(INVALIST(ALIST))
    (label  permutp_invalist) ■

    (proof invalistprop)

    ;theorem 3 (ii)

1.  (ue (chi |λalist.allp(λx.atom x,range(alist))∧injectp(alist)⊃
                   idalistp(alist ∞ invalist(alist))|) alistinduction
        (part 1(use allpfact ue: ((phi.|λx.atom x|)(x.y)(u.|range alist|)) )
        (open range injectp functp uniqueness invalist
              idalistp compalist appalist assoc)
        (use invalistsort dom_invalist compalist-lemma mode: exact>>>
    ;∀ALIST.ALLP(λX.ATOM X,RANGE(ALIST))∧INJECTP(ALIST)⊃
    ,       IDALISTP(ALIST ∞ INVALIST(ALIST))
    (label invalist-right) ■

    ;theorem 3 (iii)

2.  (assume |allp(λx.atom x,range(alist))|)

3.  (ue((alist.|invalist(alist)|)(alist1.|alist|)(za.xa)(z.y)) compalist-lemma
        (use * invalistsort range-invalist mode: exact))
    ;¬MEMBER(XA,DOM(ALIST))⊃INVALIST(ALIST) ∞ ((XA.Y).ALIST)=INVALIST(ALIST) ∞ ALIST

4.  (ci -2)
    ;ALLP(λX.ATOM X,RANGE(ALIST))⊃
```

```
;(¬MEMBER(XA,DOM(ALIST))⊃INVALIST(ALIST) ∞ ((XA.Y).ALIST)=INVALIST(ALIST) ∞ ALIST)
```

```
5.  (ue (chi |λalist.allp(λx.atom x,range(alist))∧injectp(alist)⊃
                    idalistp(invalist(alist) ∞ alist)|) alistinduction
       (part 1 (open  allp range injectp functp uniqueness
                      invalist compalist appalist assoc idalistp)
       invalistsort (use range-invalist mode: exact) (use * mode: always)))
    ;∀ALIST.ALLP(λX.ATOM X,RANGE(ALIST))∧INJECTP(ALIST)⊃
    ;        IDALISTP(INVALIST(ALIST) ∞ ALIST)
    (label invalist-left) ■
```

8.27. file PERMP: Functions Represented by Lists, Using Predicates.

```
;definitions of composition,identity, inverse as predicates
(proof comp_pred)

;composition of functions

(decl (comp) (type: |ground⊗ground⊗ground→truthval|) (syntype: constant)
      (bindingpower: 930))

(define comp |∀u v w.comp(u,v,w)≡
                    length u=length w∧(∀n.n<length u⊃nth(u,n)=nth(v,nth(w,n)))|)
(label compdef)

;the identity function

(decl (id) (type: |ground→truthval|))
(defax id |∀u.id(u)≡(∀n.n<length u⊃nth(u,n)=n)|)
(label id_def)

;the inverse of a function

(decl (inv) (type: |ground⊗ground→truthval|))
(defax inv |∀u v.inv(u,v)≡(∀n.n<length u⊃nth(u,n)=fstposition(v,n))|)
(label invdef)
```

8.27.1. Composition of Permutations is a Permutation.

```
    (proof comp_perm)

1.  (assume |perm(v)|)
    (label cp_pm1)

2.  (assume |perm(w)|)
    (label cp_pm2)

3.  (assume |length v=length w|)
    (label cp_pm3)

    (assume |comp(u, v,w)|)
    (label cp_pm4)

5.  (rw cp_pm1 (open perm into onto))
    (label cp_pm5)
    ;(∀N.N<LENGTH V⊃NATNUM(NTH(V,N))∧NTH(V,N)<LENGTH V)∧
    ;(∀N.N<LENGTH V⊃MEMBER(N,V))
```

```
6.  (rw cp_pm2 (open perm into onto))
    (label cp_pm6)
    ;(∀N.N<LENGTH W⊃NATNUM(NTH(W,N))∧NTH(W,N)<LENGTH W)∧
    ;(∀N.N<LENGTH W⊃MEMBER(N,W))

7.  (rw cp_pm4 (open comp ))
    (label cp_pm7)
    ;LENGTH U=LENGTH W∧(∀N.N<LENGTH U⊃NTH(U,N)=APPL(V,NTH(W,N)))

8.  (assume |m<length(u)|)
    (label cp_pm8)

9.  (rw * (use cp_pm7 mode: always))
    (label cp_pm9)
    ;M<LENGTH W

10. (derive |natnum(nth(w,m))∧nth(w,m)<length v| (cp_pm6 *)
           (use cp_pm3 mode: exact))

11. (trw |natnum(nth(v,nth(w,m)))∧nth(v,nth(w,m))<length v|(* cp_pm5))
    (label cp_pm10)

12. (derive |nth(u,m)=nth(v,nth(w,m))| (cp_pm7 cp_pm8)
           (open appl)  (use -2))

13. (rw cp_pm10 (use * mode: exact direction: reverse>)
    ;NATNUM(NTH(U,M))∧NTH(U,M)<LENGTH V
    (label cp_pm11)

14. (trw /length u=length v| (use cp_pm7 cp_pm3 mode: always))
    ;LENGTH U=LENGTH V

15. (rw cp_pm11  (use * mode: exact direction: reverse>>
    ;NATNUM(NTH(U,M))∧NTH(U,M)<LENGTH U
    ;deps: (CP_PM1 CP_PM2 CP_PM3 CP_PM4 CP_PM8)

16. (ci cp_pm8)
    ;M<LENGTH U⊃NATNUM(NTH(U,M))∧NTH(U,M)<LENGTH U

17. (trw lint0 u|(open into) * )
    (label cp_into)
    ;INTO(U)
    ;deps: (CP_PM1 CP_PM2 CP_PM3 CP_PM4)

18. (rw cp_pm9 (use cp_pm3 mode: exact direction: reverse>>
    ;M<LENGTH V

19. (trw |member(m,v)| (* cp_pm5))
    ;MEMBER(M,V)
    (label cp_pm20)

20. (derive |∃j.j<length(v)∧nth(v,j)=m| (* member_nth))
    (label cp_pm21)
    ;deps: (CP_PM1 CP_PM3 CP_PM4 CP_PM8)

21. (define jv |jv<length(v)∧nth(v,jv)=m| * )
    (label cp_pm22)

22. (rw * (use cp_pm3 mode: exact))
    ;JV<LENGTH W∧NTH(V,JV)=M

23. (trw |member(jv,w)| (* cp_pm6))
    ;MEMBER(JV,W)

24. (derive |∃k.k<length(w)∧nth(w,k)=jv| (* member_nth))
```

```
;deps: (CP_PM1 CP_PM2 CP_PM3 CP_PM4 CP_PM8)

25. (define kv|kv<length(w)∧nth(w,kv)=jv| * )
    (label cp_pm23)

26. (rw cp_pm22 (use * mode: always direction: reverse>>
    ;NTH(W,KV)<LENGTH V∧NTH(V,NTH(W,KV))=M
    (label cp_pm24)

27. (trw |kv<length(u)| cp_pm23 (use cp_pm7 mode: always))
    ;KV<LENGTH U
    (label cp_pm25)

28. (trw lnatnum nth(w ,kv)I cp_pm23)
    ;NATNUM(NTH(W,KV))

29. (derive |nth(u,kv)=nth(v,nth(w,kv))| (cp_pm7 cp_pm25)
            (open appl)(use *))

30. (rw * (use cp_pm24 mode: always))
    ;NTH(U,KV)=M

31. (derive |member(m,u)| nthmember
            cp_pm25 (use * mode: exact direction: reverse>>
    ;deps: (CP_PM1 CP_PM2 CP_PM3 CP_PM4 CP_PM8)

32. (ci cp_pm8)
    ;M<LENGTH U⊃MEMBER(M,U)
    (label cp-onto)

33. (trw lperm ul(open perm onto) cp_into cp-onto)
    ;PERM(U)
    ;deps: (CP_PM1 CP_PM2 CP_PM3 CP_PM4)

34. (ci (cp_pm1 cp_pm2 cp_pm3 cp_pm4))
    ;PERM(V)∧PERM(W)∧LENGTH V=LENGTH W∧COMP(U,V,W)⊃PERM(U)
    (label perm_composition) ■
```

Composition of functions is unique:

```
35. (trw |comp(u,v,w)∧comp(u1,v,w)⊃u=u1| (open comp) extensionality)
    ;COMP(U,V,W)∧COMP(U1,V,W)⊃U=U1
    (label comp_uniqueness)     ■
```

8.27.2. Composition is Associative.

```
(proof comp_associative)

1. (assume |into(w3)|)
   (label ca1)

2. (assume (length w2=length w3|)
   (label ca2)

3. (assume |comp(v,w1,w2) I>
   (label ca3)

   (assume lcomp(u,v,w3)|)
   (label ca4)

5. (assume|comp(v1,w2,w3)|)
```

```
    (label ca5)

6.  (assume|comp(u1,w1,v1)|)
    (label ca6)

7.  (assume |n<length u|)
    (label ca7)

8.  (rw ca4 (open comp))
    ;LENGTH U=LENGTH W3∧(∀N.N<LENGTH U⊃NTH(U,N)=NTH(V,NTH(W3,N)))
    (label ca8)
    ;deps: (CA4)

9.  (derive |n<length(w3) I (ca7 ca8))
    (label ca9)
    ;deps: (CA4 CA7)

10. (derive |nth(u,n)=nth(v,nth(w3,n))| (ca7 ca8))
    (label ca10)
    ;deps: (CA4 CA7)

11. (rw ca1 (open into>>
    ;∀N.N<LENGTH W3⊃NATNUM(NTH(W3,N))∧NTH(W3,N)<LENGTH W3
    ;deps: (CA1)

12. (derive |natnum(nth(w3,n))∧nth(w3,n)<length(w2)| (ca9 * ca2))
    (label call)
    ;deps: (CA1 CA2 CA4 CA7)

13. (rw ca3 (open comp))
    ;LENGTH V=LENGTH W2∧(∀N.N<LENGTH V⊃NTH(V,N)=NTH(W1,NTH(W2,N)))
    (label ca12)
    ;deps: (CA3)

14. (derive |nth(w3,n)<length(v)| (call ca12))
    (label ca13)
    ;deps: (CA1 CA2 CA3 CA4 CA7)

15. (derive |∀n.n<length(v)⊃nth(v,n)=nth(w1,nth(w2,n))| ca12)

16. (ue (n |nth(w3,n)|) * call ca13)
    ;NTH(V,NTH(W3,N))=NTH(W1,NTH(W2,NTH(W3,N)))
    ;deps: (CA1 CA2 CA3 CA4 CA7)

17.(rw ca10 (use * mode: exact))
    ;NTH(U,N)=NTH(W1,NTH(W2,NTH(W3,N)))
    (label ca14)
    ;deps: (CA1 CA2 CA3 CA4 CA7)

18.(rw ca5 (open comp))
    (label ca20)
    ;LENGTH V1=LENGTH W3∧(∀N.N<LENGTH V1⊃NTH(V1,N)=NTH(W2,NTH(W3,N)))

19. (derive |nth(v1,n)=nth(w2,nth(w3,n))| (ca9 ca20))
    (label ca21)
    ;deps: (CA4 CA5 CA7)

20.(rw ca6 (open comp))
    ;LENGTH U1=LENGTH V1∧(∀N.N<LENGTH U1⊃NTH(U1,N)=NTH(W1,NTH(V1,N)))
    (label ca22)
    ;deps: (CA6)

2 . (rw ca9 (use ca20 ca22 mode: always direction: reverse>>
    ;N<LENGTH U1
    ;deps: (CA4 CA5 CA6 CA7)
```

```
22. (derive |nth(u1,n)=nth(w1,nth(v1,n))| (ca22 *))
    ;deps: (CA4 CA5 CA6 CA7)

23. (rw * (use ca21 mode: exact))
    ;NTH(U1,N)=NTH(W1,NTH(W2,NTH(W3,N)))
    (label ca23)
    ;deps: (CA4 CA5 CA6 CA7)

24. (rw ca14 (use ca23 mode: exact direction: reverse>>
    ;NTH(U,N)=NTH(U1,N)
    ;deps: (CA1 CA2 CA3 CA4 CA5 CA6 CA7)

25. (ci ca7)
    ;N<LENGTH U⊃NTH(U,N)=NTH(U1,N)
    (label ca24)
    ;deps: (CA1 CA2 CA3 CA4 CA5 CA6)

26. (trw |length u = length u1| (use ca8 ca22 mode: always)
          (use ca20 mode: always direction: reverse)>
    ;LENGTH U=LENGTH U1
    ;deps: (CA4 CA5 CA6)

27. (ue ((u.u)(v.u1)) extensionality ca24 * )
    ;U=U1
    ;deps: (CA1 CA2 CA3 CA4 CA5 CA6)

28. (ci (ca1 ca2 ca3 ca4 ca5 ca6))
    ;INTO(W3)∧LENGTH W2=LENGTH W3∧
    ;COMP(V,W1,W2)∧COMP(U,V,W3)∧
    ;COMP(V1,W2,W3)∧COMP(U1,W1,V1)⊃U=U1
    (label associativity_pred)      ■
```

8.27.3. Using Predicates: Identity.

```
    ;id implies perm
    (proof idperm)

1.  (trw |id(u)⊃into(u)| (open id into>>
    ;ID(U)⊃INTO(U)
    (label p_i1)

2.  (assume |id(u)|)
    (label p_i2)

3.  (rw * (open id))
    ;∀N.N<LENGTH U⊃NTH(U,N)=N
    (label p_i3)

4.  (assume |n<length u|)
    (label p_i4)

5.  (derive |member(nth(u,n),u)| (* nthmember))

6.  (derive |member(n,u)| (* p_i4 p_i3))

7.  (ci p_i4)
    ;N<LENGTH U⊃MEMBER(N,U)

8.  (derive |perm u| (p_i1 p_i2 *) (open perm onto))

9.  (ci p_i2)
    ;ID(U)⊃PERM(U)
```

```
     (label  id_perm) ■

     ;Theorem 2 (ii) (id right)

     (proof  identity-right)

1.   (assume |id(u)|)
     (label id_r1)

2.   (assume |comp(v,w,u)|)
     (label id_r2)

3.   (assume llength w=length u|)
     (label id_r3)

4.   (rw id_r1 (open id))
     ;∀N.N<LENGTH U⊃NTH(U,N)=N
     (label id_r4)

5.   (rw id_r2 (open comp))
     ;LENGTH V=LENGTH U∧(∀N.N<LENGTH U⊃NTH(V,N)=NTH(W,NTH(U,N)))
     (label id_r5)

6.   (rw * (use id_r4 mode: always))
     ;LENGTH V=LENGTH U∧(∀N.N<LENGTH U⊃NTH(V,N)=NTH(W,N))
     (label id_r6)

7.   (trw llength v=length w|(use id_r3 id_r5 mode: always))
     ;LENGTH V=LENGTH W

8.   (derive |v=w|(extensionality id_r6 *))

9.   (ci (id_r1 id_r2 id_r3))
     ;ID(U)∧COMP(V,W,U)∧LENGTH W=LENGTH U⊃V=W
     (label  id-right) ■

     ;Theorem 2 (iii) (id left)

     (proof  identity-left)

1.   (assume |id(u)|)
     (label id_l1)

2.   (assume lperm w I )
     (label id_l2)

3.   (assume llength w=length u|)
     (label id_l3)

4.   (assume |comp(v,u,w)|)
     (label id_l4)

5.   (rw id-11 (open id))
     ;∀N.N<LENGTH U⊃NTH(U,N)=N
     (label id_l5)

6.   (rw id-14 (open comp))
     ;LENGTH V=LENGTH W∧(∀N.N<LENGTH V⊃NTH(V,N)=NTH(U,NTH(W,N)))
     (label id_l6)

7.   (rw id-12 (open perm onto into>>
     ;(∀N.N<LENGTH W⊃NATNUM(NTH(W,N))∧NTH(W,N)<LENGTH W)∧
     ;(∀N.N<LENGTH W⊃MEMBER(N,W))
     (label id_l7)

8.   (trw |∀m.m<length u⊃natnum(nth(w,m))∧nth(w,m)<length u| id-17
```

```
        (use id-13 mode: exact direction: reverse>>
    ;∀M.M<LENGTH U⊃NATNUM(NTH(W,M))∧NTH(W,M)<LENGTH U
    (label id_18)

9.  (trw |∀m.m<length u⊃nth(u,nth(w,m))=nth(w,m)| id-15 * )
    ;∀M.M<LENGTH U⊃NTH(U,NTH(W,M))=NTH(W,M)
    (label id_19)

10. (assume |m<length v|)
    (label id-1101

11. (trw |m<length u| *
        (use id_13 id-16 mode: exact direction: reverse>>
    ;M<LENGTH U
    (label id_111)

12. (derive |nth(u,nth(w,m))=nth(w,m)| (id-19 id_111))

13. (derive |nth(v,m)=nth(w,m) (id-16 id-1101
          (use * mode: exact direction: reverse>>

14. (ci id_110)
    ;M<LENGTH V⊃NTH(V,M)=NTH(W,M)

15. (derive |w=v|(extensionality id-16 *))

16. (ci (id_11 id-12 id-13 id_14))
    ;ID(U)∧PERM(W)∧LENGTH W=LENGTH U∧COMP(V,U,W)⊃W=V
    (label id-left) ■
```

8.27.4. Using Predicates: the Inverse Permutation Theorem.

Theorem 3 (i) (*Inv Perm*)

```
VU V.PERM(U)∧INV(V,U)∧LENGTH V=LENGTH U⊃PERM(V)
```

Part 1: inv implies into

```
    (proof inv_into)

1.  (assume |perm(u)|)
    (label ii1)

2.  (assume |inv(v,u)|)
    (label ii2)

3.  (assume |length v=length u|)
    (label ii3)

4.  (rw ii1 (open perm into onto> )
    (label ii4)
    ;(∀N.N<LENGTH U⊃NATNUM(NTH(U,N))∧NTH(U,N)<LENGTH U)∧
    ;(∀N.N<LENGTH U⊃MEMBER(N,U))

5.  (rw ii2 (open inv))
    (label ii5)
    ;∀N.N<LENGTH V⊃NTH(V,N)=FSTPOSITION(U,N)

6.  (assume |m<length v|)
    (label ii6)

7.  (derive |nth(v,m)=fstposition(u,m)| (ii5 ii6))
```

```
    (label ii7)

8.  (rw ii6 (use ii3 mode: exact))
    ;M<LENGTH U

9.  (derive |member(m,u)| (* ii4))
    ;MEMBER(M,U)

10. (trw |natnum fstposition(u,m)∧fstposition(u,m)<length u|
         (use pos_length * mode: always)
         (use posfacts ue: ((u.u)(y.m)) ))
    ;NATNUM(FSTPOSITION(U,M))∧FSTPOSITION(U,M)<LENGTH U

11. (rw * (use ii3 ii7 mode: exact direction: reverse>>
    ;NATNUM(NTH(V,M))∧NTH(V,M)<LENGTH V
    ;deps: (111 II2 II3 II6)

12. (ci ii6)
    ;M<LENGTH V⊃NATNUM(NTH(V,M))∧NTH(V,M)<LENGTH V

13. (trw lint0 v| (open into) * )
    ;INTO(V)
    ;deps: (II1 112 II3)

14. (ci (ii1 ii2 ii3))
    ;PERM(U)∧INV(V,U)∧LENGTH V=LENGTH U⊃INTO(V)
    (label inv_into)      ■
```

Part 2. inv implies perm:

```
    (proof inv_onto)

1.  (assume lperm u|)
    (label io1)

2.  (assume |inv(v,u)|)
    (label io2)

3.  (assume |length v=length u|)
    (label io3)

4.  (rw iol (open perm into onto> )
    ;(∀N.N<LENGTH U⊃NATNUM(NTH(U,N))∧NTH(U,N)<LENGTH U)∧
    ;(∀N.N<LENGTH U⊃MEMBER(N,U))
    (label io4)

5.  (rw io2 (open inv))
    ;∀N.N<LENGTH V⊃NTH(V,N)=FSTPOSITION(U,N)
    (label io5)

6.  (derive |∀n.n<length u⊃fstposition(u,nth(u,n))=n|
            (fstposition_nth perm_injectivity uniqueness-injectivity iol io4))
    ;deps: (I01)
    (label io6)

7.  (assume |n<length v|)
    (label io7)

8.  (rw * (use io3 mode: exact))
    ;N<LENGTH U
    (label io8)

9.  (derive |natnum(nth(u,n))∧nth(u,n)<length v| (io3 io4 io8))
    (label io9)
    ;deps: (I01 103 I07)
```

We can use the fact that v is the inverse of u...

```
10. (trw |nth(v,nth(u,n))=fstposition(u,nth(u,n))|(io5 *))
    ;NTH(V,NTH(U,N))=FSTPOSITION(U,NTH(U,N))
    (label io10)
    ;deps: (101 102 103 107)
```

...the lemma *Fstposition Nth*...

```
11. (ra * (use io6 io8 mode: exact))
    ;NTH(V,NTH(U,N))=N
    (label ioll)
    ;deps: (101 102 103 107)
```

...the lemma *Nthmember*...

```
12. (trw |member(nth(v,nth(u,n)),v)| (nthmember io9))
    ;MEMBER(NTH(V,NTH(U,N)),V)
    ;deps: (101 103 107)

13. (rw * (use ioll mode: exact))
    ;MEMBER(N,V)
    ;deps: (101 102 103 107)
```

. ..and obtain the second condition for *ontoness*.

```
14. (ci io7)
    ;N<LENGTH V⊃MEMBER(N,V)
    ;deps: (101 102 103)

15. (derive linto v| (inv_into iol io2 io3))
    ;deps: (101 102 103)

16. (tra lperm v| (open perm onto) -2 * )
    ;PERM(V)
    ;deps: (101 102 103)

17. (ci (iol io2 io3))
    ;PERM(U)∧INV(V,U)∧LENGTH V=LENGTH U⊃PERM(V)
    (label inv_perm)      ■
```

8.27.5. Using Predicates: the Right Inverse Theorem.

```
    ;the theorem right inverse
    (proof inverse-right)

1.  (assume lperm w|)
    (label invrl)

2.  (assume |inv(u,w)|)
    (label invr2)

3.  (assume |length u=length w|)
    (label invr3)

4.  (assume |comp(v,w,u) |>
    (label invr4)

5.  (ra invrl (open perm onto into))
    ;(∀N.N<LENGTH W⊃NATNUM(NTH(W,N))∧NTH(W,N)<LENGTH W)∧
    ;(∀N.N<LENGTH W⊃MEMBER(N,W))
```

```
    (label invr5)

6.  (rw invr2 (open inv))
    ;∀N.N<LENGTH U⊃NTH(U,N)=FSTPOSITION(W,N)
    (label invr6)

7.  (rw invr4 (open comp))
    ;LENGTH V=LENGTH U∧(∀N.N<LENGTH U⊃NTH(V,N)=NTH(W,NTH(U,N)))
  . (label invr7)

8.  (assume |m<length v|)
    (label invr8)

9.  (rw * (use invr7 mode: exact))
    ;M<LENGTH U
    (label invr9)

10. (trw |nth(v,m)=nth(w,fstposition(w,m))| (invr7 *)
         (use invr6 mode: always direction: reverse>>
    ;NTH(V,M)=NTH(W,FSTPOSITION(W,M))
    (label invr10)

11. (rw invr9 (use invr3 mode: exact))
    ;M<LENGTH W

12. (derive |member(m,w)| (invr5 *))

    ;labels: NTH_FSTPOSITION
    ;∀U N.MEMBER(N,U)⊃NTH(U,FSTPOSITION(U,N))=N

13. (rw invr10 (use nth_fstposition * mode: always))
    ;NTH(V,M)=M

14. (ci invr8)
    ;M<LENGTH V⊃NTH(V,M)=M

15. (trw |id(v)| (open id) * )
    ;ID(V)

16. (ci (invr1 invr2 invr4 invr3))
    ;PERM(W)∧INV(U,W)∧COMP(V,W,U)∧LENGTH U=LENGTH W⊃ID(V)
    (label inv_right) ∎
```

8.27.6. Using Predicates: the Left Inverse Theorem.

```
    (proof compose-inverse-left)

-1. (assume |perm(w)|)
    (label invl_1)

2.  (assume |inv(u,w)|>
    (label invl_2)

3.  (assume |comp(v,u,w)|)
    (label invl_3)

4.  (assume |length(w)=length(u)|)
    (label invl_4)

5.  (rw invl_2 (open inv))
    ;∀N.N<LENGTH U⊃NTH(U,N)=FSTPOSITION(W,N)
    (label invl_5)
```

```
6.  (rw invl_1 (open perm onto into))
    ;(∀N.N<LENGTH W⊃NATNUM(NTH(W,N))∧NTH(W,N)<LENGTH W)∧
    ;(∀N.N<LENGTH W⊃MEMBER(N,W))
    (label invl_6)
    ;deps: (INVL_1)

7.  (rw invl_3 (open comp))
    ;LENGTH V=LENGTH W∧(∀N.N<LENGTH V⊃NTH(V,N)=NTH(U,NTH(W,N)))
    (label invl_7)

8.  (derive |∀n.n<length w⊃fstposition(w,nth(w,n))=n|
            (fstposition-nth perm_injectivity uniqueness-injectivity
             invl_1 invl_6))
    (label invl_8)
    ;deps: (INVL_1)

9.  (rw invl_6 (use invl_4 mode: exact))
    ;(∀N.N<LENGTH U⊃NATNUM(NTH(W,N))∧NTH(W,N)<LENGTH U)∧
    ;(∀N.N<LENGTH U⊃MEMBER(N,W))
    (label invl_9)
    ;deps: (INVL_1 INVL_4)

10. (assume |n<length v|)
    (label invl_10)

11. (rw * (use invl_7 mode: always))
    ;N<LENGTH W
    (label invl_11)
    ;deps: (INVL_3 INVL_10)

12. (rw * invl_4)
    ;N<LENGTH U
    (label invl_12)
    ;deps: (INVL_3 INVL_4 INVL_10)

13. (derive |natnum(nth(w,n))∧nth(w,n)<length u|(invl_9 *))
    (label invl_13)
    ;deps: (INVL_1 INVL_3 INVL_4 INVL_10)

14. (derive |NTH(V,N)=NTH(U,NTH(W,N))| (invl_7 invl_10))
    (label invl_14)
    ;deps: (INVL_3 INVL_10)

15. (rw invl_14 (use invl_5 ue: ((n.|nth(w,n)|)) invl_13 mode: exact))
    ;NTH(V,N)=FSTPOSITION(W,NTH(W,N))
    (label invl_15)
    ;deps: (INVL_1 INVL_2 INVL_3 INVL_4 INVL_10)

    ;want to apply the lemma fstposition-nth

16. (rw invl_15 (use invl_8 invl_11 mode: always))
    ;NTH(V,N)=N
    ;deps: (INVL_1 INVL_2 INVL_3 INVL_4 INVL_10)

    ;and so V is the identity function

17. (ci invl_10)
    ;N<LENGTH V⊃NTH(V,N)=N
    ;deps: (INVL_1 INVL_2 INVL_3 INVL_4)

18. (trw lid v| (open id) * )
    ;ID(V)
    ;deps: (INVL_1 INVL_2 INVL_3 INVL_4)

19. (ci (invl_1 invl_2 invl_3 invl_4))
```

```
;PERM(W)∧INV(U,W)∧COMP(V,U,W)∧LENGTH W=LENGTH U⊃ID(V)
(label inverse-left) ■
```

8.28. file PERMF: Functions Represented by Lists, Using Functions.

```
;definitions of composition, identity and inverse as functions.
(proof comp_fnct)

1.  (decl def-appl (type: |@u•@u→truthval|))
2.  (define def-appl |∀uv.def_appl(v,u)≡allp(λx.natnum(x)∧x<length(v),u)|)
    (label def_appl_fact)

    ;composition of functions:

3.  (decl (compose) (infixname: |•|) (type: |ground•ground→ground|)
    (syntype: constant)(bindingpower: 930))

4.  (define compose |∀uv x.(u•nil)=nil∧
                            (u•(x.v))=(nth(u,x)).(u•v)|listinductiondef)
    (label composedef)

    ;the identity function:

5.  (decl (ident1) (type: |ground•ground→ground|))
6.  (defax ident1 |∀x u n i.ident1(i,0)=nil∧
                             ident1(i,n')=i.ident1(i',n)|)
    (label identdef1)

7.  (decl (ident) (type: |ground→ground|))
8.  (define ident |∀n.ident(n)=ident1(0,n)|)
    (label identdef )

    ;the inverse of a function:

9.  (decl (invers1) (type: |ground•ground•ground→ground|))
10. (defax invers1
       |∀u i n.invers1(u,i,0)=nil∧invers1(nil,i,n)=nil∧
               invers1(u,i,n')=if null(fstposition(u,i))
                                 then nil
                                 else fstposition(u,i).invers1(u,i',n)|)
    (label inversdef1)

11. (decl (inverse) (type: |ground→ground|))
12. (define inverse |∀u.inverse(u)=invers1(u,0,length(u))|)
    (label inversdef)
```

8.29. Condition for Definiteness and Sorts of the Functions.

```
    (proof def_appl_condition)

1.  (assume lint0 u|)

2.  (assume llength u≤length v|)

3.  (rw -2 (open into))
    ;∀N.N<LENGTH U⊃NATNUM(NTH(U,N))∧NTH(U,N)<LENGTH U

4.  (trw |∀n.n<length u⊃natnum nth(u,n)∧nth(u,n)<length v| *
```

```
    (less_lesseq_fact1 -2))
;∀N.N<LENGTH U⊃NATNUM(NTH(U,N))∧NTH(U,N)<LENGTH V

(ue ((phi1.|λx.natnum x∧x<length v|)(u.u)) nth_allp * )
;ALLP(λX.NATNUM(X)∧X<LENGTH V,U)
```

```
6.  (trw |def_appl(v,u)| (open def_appl) * )
    ;DEF_APPL(V,U)

7.  (ci (-6 -5))
    ;INTO(U)∧LENGTH U≤LENGTH V⊃DEF_APPL(V,U)
    (label def_appl_condition)     ■

    ;check sorts

    ;compose:

8.  (ue (phi |λu.def_appl(v,u)⊃listp v•u|) listinduction
        (part 1 (open def-appl allp compose )))
    ;∀U.DEF_APPL(V,U)⊃LISTP V•U
    (label sortcomp) (label simpinfo)     ■

    ;ident:

9.  (ue (a |λn.∀m.listp ident1(m,n)|) proof-by-induction
        (open ident1))
    ;∀N M.LISTP IDENT1(M,N)
    (label ident_sort1) (label simpinfo)     ■

10. (trw |∀n.listp ident(n)| (open ident) * )
    ;∀N.LISTP IDENT(N)
    (label ident_sort) (label simpinfo)     ■

    ;inverse

11. (ue (a |λn.∀i.listp invers1(u,i,n)|)
        proof-by-induction
        (open invers1) posfacts)
    ;∀N I.LISTP INVERS1(U,I,N)
    (label invers_sort1) (label simpinfo)     ■

12. (trw |listp inverse(u)| (open inverse) * )
    ;LISTP INVERSE(U)
    (label inverse-sort) (label simpinfo)     ■
```

8.30. Length Compose.

```
    ;length compose

    (proof length-compose)

    (assume|def_appl(w,u)|)
    (label l_c_1)

2.  (rw * (open def_appl))
    (label l-c-2)
    ;ALLP(λX.NATNUM(X)∧X<LENGTH W,U)

3.  (assume|n<length(u)|))
    (label l-c-3)

4.  (ue ((u.u)(x.|nth(u,n)|)(phi1.|λx.natnum(x)∧x<length(w)|))
```

```
    allp_elimination
    nthmember sexp-nth l_c_3 l-c-2)
;NATNUM(NTH(U,N))∧NTH(U,N)<LENGTH W
(label l-c-4)

5. (trw |sexp(nth(w,nth(u,n)))| sexp-nth l_c_4)
;SEXP NTH(W,NTH(U,N))
(label l_c_sort1)

6: (ci l-c-3)
;N<LENGTH U⊃SEXP NTH(W,NTH(U,N))
(label l_c_7)

7. (derive |allp(λx.natnum(x)∧x<length w,nthcdr(u,n'))|
(allp_nthcdr l-c-2))
;ALLP(λX.NATNUM(X)∧X<LENGTH W,NTHCDR(U,N'))

8. (derive |listp(w•nthcdr(u,n'))| (* sortcomp))
(label l_c_sort2)

9. (ci l-c-3)
;N<LENGTH U⊃LISTP W•NTHCDR(U,N')
(label l_c_8)

10. (ue ((phi.|λu.length(w•u)=length(u)|)(u.u))
    nthcdr_induction
    (part 1 (open compose length )) l-c-7 l-c-8)
;LENGTH (W•U)=LENGTH U

11. (ci l_c_1)
;DEF_APPL(W,U)⊃LENGTH (W•U)=LENGTH U
(label length-compose) ■
```

8.30.1. Length Ident.

```
1. (ue (a |λn.∀m.length ident1(m,n)=n|)
    proof-by-induction
    (open ident1))
;∀N M.LENGTH (IDENT1(M,N))=N
(label length_ident1) (label simpinfo)

2. (trw |∀N.LENGTH (IDENT(N))=N| * (open ident))
(label length_ident) (label simpinfo) ■
```

8.30.2. Length Inverse.

```
(proof lengthinverse)

1. (assume lperm(u)|)
(label li1)

2. (rw li1 (open perm onto into))
;(∀N.N<LENGTH U⊃NATNUM(NTH(U,N))∧NTH(U,N)<LENGTH U)∧
;(∀N.N<LENGTH U⊃MEMBER(N,U))
(label li2)

3. (ue ((u.|u|)(y.|n|)) posfacts)
;(NULL FSTPOSITION(U,N)⊃¬MEMBER(N,U))∧
```

```
;(MEMBER(N,U)⊃NATNUM(FSTPOSITION(U,N)))

4.  (derive |n<length u⊃¬null fstposition(u,n)| (3 li2))
    (label 1i3)

5.  (ue ((m.|n|) (n.|length u|)) minusfactll
        (part 1 (use less-lesseqsucc mode: exact)))
    ;N'≤LENGTH U⊃LENGTH U-N'<LENGTH U

6.  (derive |n'≤length u⊃¬null fstposition(u,length u-n')| (5 li3))
    (label li4)

7.  (trw |n'≤length u⊃(length u-n')'=length u-n|
         (use minusfact10)
         (use less-lesseqsucc mode: exact direction: reverse>>
    ;N'≤LENGTH U⊃(LENGTH U-N')'=LENGTH U-N

8.  (ue (a |λn.n≤length u⊃length (invers1(u,length u-n,n))=n|)
        proof-by-induction
        (open invers1) (use succ_lesseq_lesseq) (use 7) (use li4))
    ;∀N.N≤LENGTH U⊃LENGTH (INVERS1(U,LENGTH U-N,N))=N

9.  (ue (n |length u|)* (open lesseq))
    ;LENGTH (INVERS1(U,0,LENGTH U))=LENGTH u

10. (trw llength inverse(u)=length ul(open inverse) * )
    ;LENGTH (INVERSE(U))=LENGTH U
    ;deps: (LI1)

11. (ci li1)
    ;PERM(U)⊃LENGTH (INVERSE(U))=LENGTH U
    (label lengthinverse)     ■
```

8.30.3. Compose.

```
(proof nth-compose)

1.  (ue (phi |λu.¬null(u)∧def_appl(v,u)⊃nth(v•u,0)=nth(v,nth(u,0))|)
        listinduction
        (part 1 (open compose nth def-appl allp)) )
    ;∀U.¬NULL U∧DEF_APPL(V,U)⊃CAR (V•U)=NTH(V,CAR U)
    (label a_c_base1)

2.  (ue (phi3 |λu n.def_appl(v,u)∧n<length(u)⊃nth(v•u,n)=nth(v,nth(u,n))|)
        doubleinductionl
        (part 1 (open compose def-appl allp)) a_c_base1)
    ;∀U N.DEF_APPL(V,U)∧N<LENGTH U⊃NTH(V•U,N)=NTH(V,NTH(U,N))
    (label nth-compose) ■
```

8.30.4. Compose Permutation.

Theorem 1 (i) (YE *rm Compose)*

∀U V.PERM(U)∧PERM(V)∧LENGTH U=LENGTH V⊃PERM(U⊗V)

```
    (proof per-m-compose)

1.  (assume |perm u|)
    (label pc1)

2.  (assume lperm v|)
    (label pc2)

3.  (assume |length u = length v|)
    (label pc3)

4.  (rw pc2 (open perm onto))
    (label pc4)
    ;INTO(V)∧(∀N.N<LENGTH V⊃MEMBER(N,V))
    ;deps: (PC2)

5.  (ue ((u.v)(v.u)) def_appl_condition (open lesseq) pc3 pc4)
    ;DEF_APPL(U,V)
    (label pc5)
    ;deps: (PC2 PC3)

6.  (ue ((u.v)(w.u)) length-compose pc5)
    ;LENGTH (U•V)=LENGTH V
    (label pc6)
    ;deps: (PC2 PC3)

7.  (assume |n<length(u•v)|)
    (label pc7)

8.  (rw * (use pc6 mode: exact))
    ;N<LENGTH V
    (label pc8)
    ;deps: (PC2 PC3 PC7)

9.  (rw pc2 (open perm onto into>>
    ;(∀N.N<LENGTH V⊃NATNUM(NTH(V,N))∧NTH(V,N)<LENGTH V)∧
    ;(∀N.N<LENGTH V⊃MEMBER(N,V))
    (label pc9)
    ;deps: (PC2)

10  (derive |natnum(nth(v,n))∧nth(v,n)<length u| (pc8 *)
            (use pc3 mode: exact))
    (label pc10)
    ;deps: (PC2 PC3 PC7)

11. (rw pcl (open perm onto into>>
    ;(∀N.N<LENGTH U⊃NATNUM(NTH(U,N))∧NTH(U,N)<LENGTH U)∧
    ;(∀N.N<LENGTH U⊃MEMBER(N,U))
    (label pc11)
    ;deps: (PC1)

12. (ue (n |nth(v,n)|) * pc10)
    ;NATNUM(NTH(U,NTH(V,N)))∧NTH(U,NTH(V,N))<LENGTH U∧MEMBER(NTH(V,N),U)
    (label pc12)
    ;deps: (PC1 PC2 PC3 PC7)

13. (derive |nth(u•v,n)=nth(u,nth(v,n))| (nth_compose pc5 pc8))
    (label pc13)
    ;deps: (PC2 PC3 PC7)

14. (trw lnatnum nth(u•v,n)∧nth(u•v,n)<length(u•v)| pc12
         (use pc13 pc6 mode: exact>
         (use pc3 mode: exact direction: reverse>>
    ;NATNUM(NTH(U•V,N))∧NTH(U•V,N)<LENGTH (U•V)
    ;deps: (PC1 PC2 PC3 PC7)
```

```
15. (ci pc7)
    ;N<LENGTH (U•V)⊃NATNUM(NTH(U•V,N))∧NTH(U•V,N)<LENGTH (U•V)
    ;deps: (PC1 PC2 PC3)

16. (trw |into(u•v)|* (open into) pc5)
    ;INTO(U•V)
    (label pc-into)
    ;deps: (PC1 PC2 PC3)

    ;part 2

17. (rw pc8 (use pc3 mode: exact direction: reverse>>
    ;N<LENGTH U
    (label pc20)
    ;deps: (PC2 PC3 PC7)

    ;labels: MEMBER-NTH
    ;∀U Y.MEMBER(Y,U)⊃(∃N.N<LENGTH U∧NTH(U,N)=Y)

18. (define jv |jv<length u ∧nth(u,jv)=n| (* pc11 member_nth))
    (label pc21)
    ;JV is unknown.
    ;the symbol JV is given the same declaration as J
    ;deps: (PC1 PC2 PC3 PC7)

19. (derive |jv<length v| * (use pc3 mode: exact direction: reverse>>
    ;deps: (PC1 PC2 PC3 PC7)

20. (define kv |kv<length v ∧nth(v,kv)=jv| (* pc9 member_nth))
    (label pc22)
    ;KV is unknown.
    ;the symbol KV is given the same declaration as K
    ;deps: (PC1 PC2 PC3 PC7)

    ;labels: NTH_COMPOSE
    ;∀V U N.DEF_APPL(V,U)∧N<LENGTH U⊃NTH(V•U,N)=NTH(V,NTH(U,N))

21. (ue ((v.u)(u.v)(n.kv)) nth_compose pc5
        (use * mode: always)(use pc21 mode: always))
    (label pc23)
    ;NTH(U•V,KV)=N
    ;deps: (PC1 PC2 PC3 PC7)

22. (derive |kv<length(u•v)| (pc22 pc6))
    ;deps: (PC1 PC2 PC3 PC7)

    ;labels: NTHMEMBER
    ;∀U N.N<LENGTH U⊃MEMBER(NTH(U,N),U)

23. (trw |member(nth(u•v,kv),u•v)| (nthmember pc5 *))
    ;MEMBER(NTH(U•V,KV),U•V)
    ;deps: (PC1 PC2 PC3 PC7)

24. (rw * pc23)
    ;MEMBER(N,U•V)
    ;deps: (PC1 PC2 PC3 PC7)

25. (ci pc7)
    ;N<LENGTH (U•V)⊃MEMBER(N,U•V)
    ;deps: (PC1 PC2 PC3)
    (label pc-onto)

26. (trw |perm(u•v)| (pc5 pc-into pc-onto) (open perm onto))
    ;PERM(U•V)
    ;deps: (PC1 PC2 PC3)
```

```
27. (ci (pc1 pc2 pc3))
    ;PERM(U)∧PERM(V)∧LENGTH U-LENGTH V⊃PERM(U•V)
    (label perm_compose) ■

    ;Theorem 1 (ii) (associativity of composition)

    (proof assoc_compose)

1.  (trw |def_appl(w,v)∧def_appl(v,u)⊃(w•v)•nil=w•(v•nil)|
        (open compose) sortcomp)
    (label ass_comp_base)

2.  (ue (phi |λu.def_appl(w,v)∧def_appl(v,u)⊃(w•v)•u=w•(v•u)|)
       listinduction
       (part 1#2 (open compose def_appl allp)) sortcomp ass_comp_base
       (use nth_compose ue: ((v.w)(u.v)) ) )
    ;∀U.DEF_APPL(W,V)∧DEF_APPL(V,U)⊃(W•V)•U=W•(V•U)
    (label assoc_comp)

3.  (assume |perm(v)∧perm(u)∧length(v)=length(u)∧length(w)=length(v)|)

4.  (rw * (open perm onto))
    ;INTO(V)∧(∀N.N<LENGTH U⊃MEMBER(N,V))∧
    ;INTO(U)∧(∀N.N<LENGTH U⊃MEMBER(N,U))∧
    ;LENGTH V=LENGTH UALENGTH W=LENGTH U

5.  (ue ((u.v)(v.w)) def_appl_condition * (open lesseq))
    ;DEF_APPL(W,V)

6.  (ue ((u.u)(v.v)) def_appl_condition -2 (open lesseq))
    ;DEF_APPL(V,U)

7.  (derive |(w•v)•u=w•(v•u) I (assoc_comp * -2))

8.  (ci -5)
    ;PERM(V)∧PERM(U)∧LENGTH V=LENGTH U∧LENGTH W=LENGTH U⊃(W•V)•U=W•(V•U)
    (label associativity-of-composition) ■
```

8.30.5. Identity.

```
    (proof id-main)
    ;id main

1.  (assume |n<m⊃nthcdr(ident(m),n)=ident1(n,m-n) |>
     label id_main1)

2.  (assume |n'<m|)
    (label id_main2)

3.  (derive|nthcdr(ident(m),n)=ident1(n,m-n)|
          (id_main1 id-main2 succ-less-less))
    ;deps: (ID-MAIN1 ID_MAIN2)

4.  (rw * (use minusfact10 mode: exact) (open ident1)
       (use id-main2 succ-less-less mode: exact))
    ;NTHCDR(IDENT(M),N)=N.IDENT1(N',M-N')
    ;deps: (ID_MAIN1 ID_MAIN2)

5.  (trw |nthcdr(ident m,n') |
        (use cdr_nthcdr mode: exact direction: reverse)
        (use * mode: exact))
    ;NTHCDR(IDENT(M),N')=IDENT1(N',M-N')
```

```
6.  (ci id_main2)
    ;N'<M⊃NTHCDR(IDENT(M),N')=IDENT1(N',M-N')

7.  (ci id_main1)
    ;(N<M⊃NTHCDR(IDENT(M),N)=IDENT1(N,M-N))⊃
    ;(N'<M⊃NTHCDR(IDENT(M),N')=IDENT1(N',M-N'))

8.  (ue (a |λn.n<m⊃nthcdr(ident(m),n)=ident1(n,m-n)|)
       proof_by_induction
       (part 1#1 (open minus ident)) * )
    ;∀N.N<M⊃NTHCDR(IDENT(M),N)=IDENT1(N,M-N)
    (label nthcdr_ident)

9.  (rw * (use minusfact10 mode: exact))
    ;∀N.N<M⊃NTHCDR(IDENT(M),N)=IDENT1(N,(M-N')')

10. (ue ((u.|ident m|)(n.n)) car_nthcdr (use * mode: always))
    ;N<M⊃N=NTH(IDENT(M),N)

11. (trw |∀n m.n<m⊃nth(ident m,n)=n| * )
    (label id_main)    ■

    (proof perm_ident)

    ;only ontoness requires some help

1.  (assume |n<length ident(m)|)
    (label prm_id1)

2.  (rw * (open ident))
    ;N<M
    (label prm_id2)

3.  (derive |nth(ident(m),n)=n| (* id_main))

4.  (derive |member(nth(ident m,n),ident m)|
            (nthmember prm_id1) )

5.  (rw * (use -2 mode: exact))
    ;MEMBER(N,IDENT(M))

6.  (ci prm_id1)
    ;N<M⊃MEMBER(N,IDENT(M))

7.  (trw |∀n.perm(ident n)| (open perm into onto)
         (use id_main mode: always) * )
    ;∀N.PERM(IDENT(N))
    (label   perm_ident)  ■
```

8.30.6. Right Identity.

```
    (proof identity_right)

1.  (rw perm_id (open perm onto))
    ;∀N.INTO(IDENT(N))∧(∀N1.N1<N⊃MEMBER(N1,IDENT(N)))

    ;labels: DEF_APPL_CONDITION
    ;∀U V.INTO(U)∧LENGTH U≤LENGTH V⊃DEF_APPL(V,U)

2.  (ue ((u.|ident(length u)|)(v.u))
       def_appl_condition  * (open lesseq))
```

```
;DEF_APPL(U,IDENT(LENGTH U))

;labels: NTH_COMPOSE
;∀V U N.DEF_APPL(V,U)∧N<LENGTH U⊃NTH(V•U,N)=NTH(V,NTH(U,N))

3.  (ue ((u.|ident(length u) |)(v.u)(n.n)) nth_compose *
       (use id_main mode: exact))
    ;N<LENGTH U⊃NTH(U•IDENT(LENGTH U),N)=NTH(U,N)

    ;labels: EXTENSIONALITY
    ;∀U V.LENGTH U=LENGTH V∧(∀I.I<LENGTH U⊃APPL(U,I)=APPL(V,I))⊃U=V

4.  (ue ((u.|u•ident(length u)|)(v.u)) extensionality (open appl)
       (use length_compose -2 *))
    ;U•IDENT(LENGTH U)=U
    (label identity_right)   ■
```

8.30.7. Left Identity.

```
    (proof identity_left)

1.  (assume |into u|)
    (label il_1)

2.  (ue ((u.u)(v.|ident(length u)|))
        def_appl_condition
        * (open lesseq))
    ;DEF_APPL(IDENT(LENGTH U),U)
    (label il_2)

3.  (rw il_1 (open into))
    ;∀N.N<LENGTH U⊃NATNUM(NTH(U,N))∧NTH(U,N)<LENGTH U

4.  (ue ((v.|ident(length u)|)(u.u)) nth_compose il_2 *
        (use id_main ue: ((n.|nth(u,n)|)(m.|length u|)) ))
    ;∀N.N<LENGTH U⊃NTH(IDENT(LENGTH U)•U,N)=NTH(U,N)

5.  (ue ((u.|ident(length u)•u|)(v.u)) extensionality
        (sortcomp il_2 length_compose *) (open appl))
    ;IDENT(LENGTH U)•U=U

6.  (ci il_1)
    ;INTO(U)⊃IDENT(LENGTH U)•U=U
    (label  identity_left)  ■
```

8.30.8. Inverse.

```
    (proof inverse_main)

1.  (assume |perm u|)
    (label inv_main1)

    ;check that fstposition has the proper value on the intended domain

2.  (rw inv_main1 (open perm onto))
    ;INTO(U)∧(∀N.N<LENGTH U⊃MEMBER(N,U))
    (label inv_main2)
```

```
3.  (ue ((u.u)(y.n)) posfacts)
    ;(NULL FSTPOSITION(U,N)⊃¬MEMBER(N,U))∧
    ;(MEMBER(N,U)⊃NATNUM(FSTPOSITION(U,N)))

4.  (derive |n<length u⊃¬null fstposition(u,n)| (inv_main2 *))
    (label inv_main3)

    ;prove by induction a sublemma:

5.  (assume |n<length u⊃
             nthcdr(inverse(u),n)=invers1(u,n,length u-n)|)
    (label inv_main5)

6.  (assume |n'<length u|)
    (label inv_main6)

    (derive |n<length u| (* succ_less_less))
    (label inv_main7)

8.  (derive |¬null fstposition(u,n) | (inv_main3 inv_main7))
    (label inv_main9)

9.  (rw inv_main5
        (use inv_main7 inv_main9)(open invers1)
        (use minusfact10 mode: always))
    (label inv_main10)
    ;NTHCDR(INVERSE(U),N)=FSTPOSITION(U,N).INVERS1(U,N',LENGTH U-N')
    ;deps: (INV_MAIN1 INV_MAIN5 INV_MAIN6)

    ;labels: CDR_NTHCDR
    ;∀U N.CDR NTHCDR(U,N)=NTHCDR(U,N')

10. (ue ((u.|inverse u|)(n.n)) cdr_nthcdr (use * mode: exact))
    ;INVERS1(U,N',LENGTH U-N')=NTHCDR(INVERSE(U),N')
    ;deps: (INV_MAIN1 INV_MAIN5 INV_MAIN6)

11. (ci inv_main6)
    ;N'<LENGTH U⊃INVERS1(U,N',LENGTH U-N')=NTHCDR(INVERSE(U),N')

12. (ci inv_main5)

13. (ue (a |λn.n<length u⊃nthcdr(inverse(u),n)=invers1(u,n,length u-n)|)
        proof_by_induction  (part 1#1 (open inverse minus)) * )
    ;∀N.N<LENGTH U⊃NTHCDR(INVERSE(U),N)=INVERS1(U,N,LENGTH U-N)
    ;deps: (INV_MAIN1)

    ;from this the main lemma follows:

14. (rw * (use minusfact10 mode: exact) (open invers1)
        (use inv_main3 mode: always))
    ;∀N.N<LENGTH U⊃
    ;NTHCDR(INVERSE(U),N)=FSTPOSITION(U,N).INVERS1(U,N',LENGTH U-N')
    ;deps: (INV_MAIN1)

    ;labels: CAR_NTHCDR
    ;∀U N.N<LENGTH U⊃CAR NTHCDR(U,N)=NTH(U,N)

15. (ue ((u.|inverse(u)|)(n.n)) car_nthcdr
        (use * lengthinverse inv_main1 mode: always))
    ;N<LENGTH U⊃FSTPOSITION(U,N)=NTH(INVERSE(U),N)
    ;deps: (INV_MAIN1)

16. (ci inv_main1)
    ;PERM(U)⊃(N<LENGTH U⊃FSTPOSITION(U,N)=NTH(INVERSE(U),N))

17. (derive |∀u n.perm u∧n<length u⊃nth(inverse u,n)=fstposition(u,n)| * )
```

```
(label   inv_main)  ∎
```

8.30.9. Inverse Permutation.

```
 · (proof inverse_perm)

1.  (assume |perm(u)|)
    (label inv_p1)

2.  (rw * (open perm onto))
    ;INTO(U)∧(∀N.N<LENGTH U⊃MEMBER(N,U))
    (label inv_p2)

3.  (ue ((u.u)(y.n)) posfacts)
    ;(NULL FSTPOSITION(U,N)⊃¬MEMBER(N,U))∧
    ;(MEMBER(N,U)⊃NATNUM(FSTPOSITION(U,N)))

4.  (derive |∀n.n<length u⊃
            natnum fstposition(u,n)∧fstposition(u,n)<length u|
           (inv_p2 * pos_length))
    (label inv_p3)

5.  (derive |∀n.n<length u⊃
            nth(inverse u,n)=fstposition(u,n)|
           (inv_main inv_p1))
    (label inv_p4)

6.  (rw inv_p3 (use * mode: always direction: reverse))
    ;∀N.N<LENGTH U⊃NATNUM(NTH(INVERSE(U),N))∧NTH(INVERSE(U),N)<LENGTH U

7.  (trw |into inverse(u)| *
         (open into) (use lengthinverse inv_p1 mode: exact))
    ;INTO(INVERSE(U))
    (label into_inverse)

8.  (ci inv_p1)
    ;PERM(U)⊃INTO(INVERSE(U))
    (label inv_into)

9.  (rw inv_p1 (open perm into onto) )
    ;(∀N.N<LENGTH U⊃NATNUM(NTH(U,N))∧NTH(U,N)<LENGTH U)∧
    ;(∀N.N<LENGTH U⊃MEMBER(N,U))
    (label inv_p10)

10. (derive |length inverse(u)=length u| (inv_p1 lengthinverse))
    (label inv_p11)

11. (assume |n<length inverse(u)|)
    (label inv_p12)

12. (rw * (use inv_p11 mode: exact))
    ;N<LENGTH U
    (label inv_p13)

    ;apply the main property of the inverse function...

13. (ue (n |nth(u,n)|) inv_p4 (use inv_p10 * mode: always))
    ;NTH(INVERSE(U),NTH(U,N))=FSTPOSITION(U,NTH(U,N))
    (label inv_p14)

    ;...the consequence of the Pigeon Hole principle...
```

```
14. (derive |inj u|(inv_p1 perm_injectivity))

    ;...the basic fact fstposition nth ...

15. (derive |fstposition(u,nth(u,n))=n|
            (fstposition_nth uniqueness_injectivity * inv_p10 inv_p13))

16. (rw inv_p14 (use *))
    ;NTH(INVERSE(U),NTH(U,N))=N
    (label inv_p15)

    ;...and the lemma nthmember..

17. (derive |natnum nth(u,n)∧nth(u,n)<length inverse(u)|
            (inv_p10 inv_p11 inv_p13))

18. (trw |member(nth(inverse u,nth(u,n)),inverse u)|
        (nthmember *))
    ;MEMBER(NTH(INVERSE(U),NTH(U,N)),INVERSE(U))

    ;..to  conclude:

19. (rw * (use inv_p15))
    ;MEMBER(N,INVERSE(U))
    ;deps: (INV_P1 INV_P12)

20. (ci inv_p12)
    ;N<LENGTH (INVERSE(U))⊃MEMBER(N,INVERSE(U))
    (label onto_inverse)

21. (trw |perm(inverse u) | (open perm onto)
         into_inverse onto_inverse)
    ;PERM(INVERSE(U))

22. (ci inv_p1)
    ;PERM(U)⊃PERM(INVERSE(U))
    (label   perm_inverse)   ■
```

8.30.10. Right Inverse.

Theorem 3. (ii) (*Right Inverse*)

```
∀U.PERM(U)⊃U⊗INVERSE(U)=IDENT(LENGTH(U))
```

Proof. We aim at an application of extensionality (line 12). From line 8 on, we follow the proof given in Section 6.6.2, since all the facts assumed there as definitions have been proved here as properties of our functions.

The additional fact to be proved here is that u is defined as an application on inverse(u) as domain. This follows from the fact that inverse(u) is into (line 5) and has the same length as u (line 3).

```
    (proof compose_inverse_right)

1.  (assume |perm u|)(label cir1)

2.  (rw cir1 (open perm onto))
    ;INTO(U)∧(∀N.N<LENGTH U⊃MEMBER(N,U))
    (label cir2)

    ;labels: LENGTHINVERSE
    ;PERM(U)⊃LENGTH (INVERSE(U))=LENGTH U

3.  (derive |length inverse(u)=length u| (cir1 lengthinverse))
    (label cir3)

    ;labels: PERM_INVERSE
    ;PERM(U)⊃PERM(INVERSE(U))

4.  (derive |perm inverse(u)|(perm_inverse cir1))

5.  (rw * (open perm onto))
    ;INTO(INVERSE(U))∧(∀N.N<LENGTH (INVERSE(U))⊃MEMBER(N,INVERSE(U)))

    ;labels: DEF_APPL_CONDITION
    ;∀U V.INTO(U)∧LENGTH U≤LENGTH V⊃DEF_APPL(V,U)

6.  (ue ((v.u)(u.|inverse u|))
        def_appl_condition
        (cir3 *) (open lesseq))
    ;DEF_APPL(U,INVERSE(U))
    (label cir4)

    ;labels: LENGTH_COMPOSE
    ;∀W U.DEF_APPL(W,U)⊃LENGTH (W•U)=LENGTH U

7.  (trw |length(u•inverse(u))=length ident(length u)|
         (use length_compose ue: ((w.u)(u.|inverse u|)) cir4 mode: always)
         (use cir3))
    ;LENGTH (U•INVERSE(U))=LENGTH (IDENT(LENGTH U))
    (label cir5)
```

we can apply *Nth* Compose...

```
    ;labels: NTH_COMPOSE
    ;∀V U N.DEF_APPL(V,U)∧N<LENGTH U⊃NTH(V•U,N)=NTH(V,NTH(U,N))

8.  (ue ((v.u)(u.|inverse u|))
        nth_compose cir4 cir3)
    ;∀N.N<LENGTH U⊃NTH(U•INVERSE(U),N)=NTH(U,NTH(INVERSE(U),N))
```

...Main Inv...

```
    ;labels: INV_MAIN
    ;∀U N.PERM(U)∧N<LENGTH U⊃NTH(INVERSE(U),N)=FSTPOSITION(U,N)

9.  (rw * (use inv_main cir1 mode: always))
    ;∀N.N<LENGTH U⊃NTH(U•INVERSE(U),N)=NTH(U,FSTPOSITION(U,N))
```

...Nth Fstposition...

```
    ;labels: NTH_FSTPOSITION
    ;∀U N.MEMBER(N,U)⊃NTH(U,FSTPOSITION(U,N))=N

10. (rw * (use nth_fstposition cir2 mode: always))
    ;∀N.N<LENGTH U⊃NTH(U•INVERSE(U),N)=N
```

. ..to conclude, using *Main Id*:

```
;labels: ID_MAIN
;∀M N.N<M⊃NTH(IDENT(M),N)=N

11. (trw |∀n.n<length u⊃nth(u•inverse(u),n)=nth(ident(length u),n) |
         (use * mode: always)
         (use id_main ue: ((m.|length u|)(n.n)) mode: always))
    ;∀N.N<LENGTH U⊃NTH(U•INVERSE(U),N)=NTH(IDENT(LENGTH U),N)
    (label cir6)

    ;labels: EXTENSIONALITY
    ;∀U V.LENGTH U=LENGTH V∧(∀I.I<LENGTH U⊃APPL(U,I)=APPL(V,I))⊃U=V

12. (ue ((u.|u•inverse(u)|)(v.|ident(length u)|))
        extensionality cir6
        (use cir5 mode: always)(use sortcomp cir4 mode: always))
    ;U•INVERSE(U)=IDENT(LENGTH U)

13. (ci cir1)
    ;PERM(U)⊃U•INVERSE(U)=IDENT(LENGTH U)
    (label   inverse_right)   ■
```

8.30.11. Left Inverse.

Theorem 3. (iii) (*Left Inverse*)

```
∀U.PERM(U)⊃INVERSE U•U=IDENT(LENGTH U)
```

Proof. Again we follow closely the pattern of Section 6.6.3.

```
    (proof compose_inverse_left)

1.  (assume |perm u|)
    (label cil1)

2.  (derive |length inverse(u)=length u| (lengthinverse *))
    (label cil2)

3.  (rw cil1 (open perm onto))
    ;INTO(U)∧(∀N.N<LENGTH U⊃MEMBER(N,U))
    (label cil3)

    ;labels: DEF_APPL_CONDITION
    ;∀U V.INTO(U)∧LENGTH U≤LENGTH V⊃DEF_APPL(V,U)

4.  (ue ((v.|inverse u|)(u.u)) def_appl_condition
        cil2 cil3 (open lesseq)
        (use perm_inverse cil1))
    ;DEF_APPL(INVERSE(U),U)
    (label cil4)

5.  (trw |listp inverse(u)•u| (cil4 sortcomp))
    ;LISTP INVERSE(U)•U
    (label cilsort)

6.  (derive |length(inverse(u)•u)=length ident(length u)|
            (cil4 length_compose))
    (label cil5)

7.  (assume |n<length u|)(label cil6)
```

Use the lemma *Nth Compose*...

```
;labels: NTH_COMPOSE
;∀V U N.DEF_APPL(V,U)∧N<LENGTH U⊃NTH(V•U,N)=NTH(V,NTH(U,N))

8. (ue ((v.|inverse u|)(u.u)(n.n)) nth_compose cil4 cil6)
   ;NTH(INVERSE(U)•U,N)=NTH(INVERSE(U),NTH(U,N))
   (label cil7)
```

...the main property of inverse...

```
9. (rw cil3 (open into))
   ;(∀N.N<LENGTH U⊃NATNUM(NTH(U,N))∧NTH(U,N)<LENGTH U)∧
   ;(∀N.N<LENGTH U⊃MEMBER(N,U))
   (label cil8)

   ;labels: INV_MAIN
   ;∀U N.PERM(U)∧N<LENGTH U⊃NTH(INVERSE(U),N)=FSTPOSITION(U,N)

10. (ue ((u.u)(n.|nth(u,n)|)) inv_main (use cil1 cil6 cil8 mode: always))
   ;NTH(INVERSE(U),NTH(U,N))=FSTPOSITION(U,NTH(U,N))
   (label cil9)
```

...a. consequence of the Pigeon Hole principle...

```
11. (derive |inj u| (perm_injectivity cil1))
   (label cil10)
```

...the lemma *Fstposition Nth*...

```
;labels: FSTPOSITION_NTH
;∀U N.UNIQUENESS(U)∧N<LENGTH U⊃FSTPOSITION(U,NTH(U,N))=N

12. (derive |fstposition(u,nth(u,n))=n|
          (fstposition_nth uniqueness injectivity cil10 cil6))

13. (rw cil9 (use *))
   ;NTH(INVERSE(U),NTH(U,N))=N
```

...and the main property of `ident` to conclude:

```
;labels: ID_MAIN
;∀M N.N<M⊃NTH(IDENT(M),N)=N

14. (trw |nth(inverse(u)•u,n)=nth(ident(length u),n)|
        (use cil6 cil7 * mode: always)
        (use id_main ue: ((m.|length u|)(n.n)) cil6 mode: always))
   ;NTH(INVERSE(U)•U,N)=NTH(IDENT(LENGTH U),N)

15. (ci cil6)
   ;N<LENGTH U⊃NTH(INVERSE(U)•U,N)=NTH(IDENT(LENGTH U),N)
```

Therefore:

```
;labels: EXTENSIONALITY
;∀U V.LENGTH U=LENGTH V∧(∀I.I<LENGTH U⊃APPL(U,I)=APPL(V,I))⊃U=V

16. (ue ((u.|inverse(u)•u|)(v.|ident(length u)|))
      extensionality
      cil5 * (open appl))
   ;INVERSE(U)•U=IDENT(LENGTH U)
   ;deps: (CIL1)

17. (ci cil1)
   ;PERM(U)⊃INVERSE(U)•U=IDENT(LENGTH U)
   (label inverse left)     ■
```

9. Bibliography.

R.S.Boyer and J.S.Moore [1979],
A Computational Logic, Academic Press, New York, 1979.

C.Goad [1979],
Independence of the Pigeon Hole Principle, manuscript,, Stanford.

Jussi Ketonen and Joseph S. Weening [1983]
The Language of an Interactive Proof Checker, Stanford Report STAN-CS-83-992

Jussi Ketonen and Joseph S. Weening [1984]
EKL–An Interactive Proof Checker. User's Reference Manual, Stanford Report STAN-CS-84-1006

G.Kreisel [1981],
Neglected Possibilities of Processing Assertions and Proofs Mechanically: Choice of Problems and Data, in Suppes (ed.) University-Level Computer-Assisted Instruction at Stanford: 1968-1980, IMSSS Stanford, 1981.

D.Prawitz [1965],
Natural Deduction, A Proof- Theoretical Study, Almqvist and Wiksell, Stockholm, Göteborg, Uppsala, 1965.

D.Prawitz [1968],
Hauptsatz for Higher Order Logic, Journal of Symbolic Logic, 33, **1968,** pp. 452-457.

D.Prawitz [1971],
Ideas and Results in Proof Theory, in *Proceedings of the Second Scandinavian Logic Symposium,* Oslo 1.970, Fenstad Editor, North Holland, Amsterdam, 1971, pp. 235-307.

A.S.Troelstra [1973],
Metamathematical Investigations of Intuitionistic Arithmetic and Analysis, Lecture Notes in Mathematics, 3-f-l. Springer, Berlin, Heidelberg, New York, 1973.

9.1. Index of Examples.

Example 1: Use of the rewriter NORMAL. Section 2.1.
Proof: transitivity of $\leq$.

```
∀N M K.N≤M∧M≤K⊃N≤K
```

Example *2:* Default declarations and previous declarations, Section 2.4.
Declaration of the symbol xv.

Example 3: Rewriting using only simpinfo. Section 2.6.
Proof: *Sexp Nth.*

```
∀U N.SEXP NTH(U,N)
```

Example 4: How the rewriting process reflects an informal argument. Section 2.9.
Proof: *Fstposition Nth*

```
∀U N.UNIQUENESS(U)∧N<LENGTH U⊃FSTPOSITION(U,NTH(U,N))=N
```

Example 5. Abbreviation of proofs by rewriting. How the rewriter *Trans Cond* is determined in a 'trial and error' interaction. Section 7.
Proof: **Lemma 2.10** *Mult Nthcdr,*

```
∀N A U.N<LENGTH U⊃MULT(NTHCDR(U,N),A)≤MULT(U,A)
```

Example 6: Predicate all. Definition by recursion and explicit definitions. Heuristics. Section 4.1.
Proof: *Pigeonfact*

```
∀F.(∀N.NATNUM(F(N)))⊃
   (∀N.(∀M.M<N⊃1≤F(M))∧SUM(λK.F(K),N)=N⊃(∀M.M<N⊃1=F(M)))
```

Example 7: Predicate somep. Efficiency of rewriting. Definition by recursion and explicit definitions. Section 5.2.
Proof: *Nonempty Range*

```
∀ALIST X.MEMBER(X,DOM ALIST)⊃
        (∃Y.MEMBER(Y,RANGE ALIST)∧APPALIST(X,ALIST)=Y)
```

Example 8: Heuristics of a. proof. Section 6.3.4.
Proof: **Lemma** *6.3. Lengthinverse*

```
∀U.PERM(U)⊃LENGTH (INVERSE(U))=LENGTH U
```

Example 9: Use of general lemmata and efficiency. Section 6.5.4.
Proof: **Theorem 2** (ii) (*Right Identity)*

```
∀U.U⊗IDENT(LENGTH U)=U
```

Example 10: Discussion of the formalization of an informal argument.. Section 7.
Proof: *Every surjection of finite sets of the same cardinality is an injection.*

10. Index of SIMPINFO.

The following: lines are labeled simpinfo in some proof. They are available to EKL in the execution of all proofs that use the proof in question by the command get-proofs (we the graph of file dependency at the beginning of the Appendix).

Simpinfo from file LISPAX
proof LISPAX

```
13. ∀XA.SEXP XA

14. ∀U.SEXP U

15. ∀X U.LISTP X.U

16. ∀U.¬NULL U⊃LISTP CDR U

17. ∀U.¬NULL U⊃SEXP CAR U

18. ∀X.¬ATOM X⊃SEXP CAR X

19. ∀X.¬ATOM X⊃SEXP CDR X

20. ∀X Y.SEXP X.Y

21. ∀X Y.¬ATOM X.Y

22. ∀X U.¬NULL X.U

23. ∀U.NULL U⊃U=NIL

24. ∀X Y.CAR (X.Y)=X

25. ∀X Y.CDR (X.Y)=Y

26. CAR NIL=NIL

27. CDR NIL=NIL

28. ∀U.¬NULL U⊃CAR U.CDR U=U

;labels: SIMPINFO CONS_CAR_CDR
29. ∀X.¬ATOM X⊃CAR X.CDR X=X

43. LIST(())=NIL

44. ∀LST.LISTP LIST(LST)

;labels: SIMPINFO LISTDEF
45. ∀X LST.LIST(X,LST)=X.LIST(LST)

;labels: SIMPINFO APPENDEF
47. ∀X U V.NIL*V=V∧X.U*V=X.(U*V)

;labels: SIMPINFO LISTAPPEND
48. ∀U V.LISTP U*V

49. ∀U.U*NIL=U

50. ∀X V.X.NIL*V=X.V

56. ∀ALIST.LISTP ALIST
```

```
;labels: SIMPINFO ALISTDEF
58. ∀XA Y ALIST.ALISTP NIL∧ALISTP (XA.Y).ALIST

61. ∀X ALIST.SEXP ASSOC(X,ALIST)

66. ∀U.SEXP CAR U

67. ∀U.LISTP CDR U
```

Simpinfo from file SET

proof SETS

```
3.  ∀X.URELEMENT X

4.  ∀XV.SEXP(XV)
```

Simpinfo from file NATNUM

proof NATNUM

```
10. ∀N.NATNUM(N')

;labels: SIMPINFO PRED_DEF
19. ∀N.PRED(N')=N

20. ∀N.NATNUM(PRED(N))

;labels: SIMPINFO PLUSFACTS PLUSDEF
21. ∀N K.0+N=N∧K'+N=(K+N)'

22. ∀N M.NATNUM(N+M)

;labels: SIMPINFO PLUSFACTS
23. ∀N.N+0=N

;labels: SIMPINFO PLUSFACTS PLUSDEF1
24. ∀N.1+N=N'∧N+1=N'

;labels: SIMPINFO PLUSFACTS
25. ∀N K.N+K'=(N+K)'

;labels: SIMPINFO SUCCFACTS ZERO_NOT_SUCCESSOR
17. ∀N.¬N'=0

;labels: SIMPINFO ZEROLEAST1
9. ∀N.¬N<0

;labels: SIMPINFO SUCCFACTS SUCCESSORLESS
13. ∀N M.N'<M'≡N<M

;labels: SIMPINFO SUCCFACTS SUCCESSOREQ
14. ∀N M.N'=M'≡N=M

;labels: SIMPINFO SUCCFACTS ZEROLEAST3
16. ∀N.0<N'
```

Simpinfo from file MINUS

proof LESSEQ

```
1. ∀N.¬N=N'

;labels: SIMPINFO SUCCESSORFACTS SUCCESSORLESSEQ
4. ∀N M.N'≤M'≡N≤M

;labels: SIMPINFO ZERO_NON_LESS_SUCCESSOR
9. ∀N M.N'<M⊃¬M=0
```

proof MINUS

```
;labels: SIMPINFO MINUS_SORT
3. ∀N K.NATNUM(K-N)

;labels: SIMPINFO N_LESS_N
9. ∀N.N-N=0
```

Simpinfo from file LENGTH

proof LENGTH

```
;labels: SIMPINFO LENGTHDEF
2. ∀U X.LENGTH NIL=0∧LENGTH (X.U)=LENGTH U'

3. ∀U.NATNUM(LENGTH U)

4. ∀U.LENGTH U=0≡NULL U

;labels: SIMPINFO LENGTHADD
5. ∀U V.LENGTH (U*V)=LENGTH U+LENGTH V

6. ∀X.LENGTH (X.NIL)=1

;labels: SIMPINFO HAVE_MEMBER
8. ∀U Y.MEMBER(Y,U)⊃0<LENGTH U

;labels: SIMPINFO HAVE_MEMBER1
9. ∀U Y.MEMBER(Y,U)⊃¬NULL U
```

Simpinfo from file NTH

proof LISPAX

```
69. ∀N.¬NULL N

70. ∀N.SEXP N
```

proof NTH

```
;labels: SIMPINFO NTHDEF
2. ∀X U N.NTH(NIL,N)=NIL∧NTH(U,0)=CAR U∧NTH(X.U,N')=NTH(U,N)

;labels: SIMPINFO SEXP_NTH
3. ∀U N.SEXP NTH(U,N)
```

proof NTHCDR

```
;labels: SIMPINFO DEF
2. ∀X U N.NTHCDR(NIL,N)=NIL∧NTHCDR(U,0)=U∧NTHCDR(X.U,N')=NTHCDR(U,N)

3. ∀U N.LISTP NTHCDR(U,N)
```

proof FSTPOSITION

```
;labels: SIMPINFO POSFACTS
3. ∀U Y.(NULL FSTPOSITION(U,Y)⊃¬MEMBER(Y,U))∧
       (MEMBER(Y,U)⊃NATNUM(FSTPOSITION(U,Y)))∧
       (NULL FSTPOSITION(U,Y)∨NATNUM(FSTPOSITION(U,Y)))

;labels: SIMPINFO SORTPOS
4. ∀U Y.SEXP FSTPOSITION(U,Y)
```

Simpinfo from file APPL

proof ALISTFACTS

```
;labels: SIMPINFO DOMSORT
1. ∀ALIST.LISTP DOM(ALIST)

;labels: SIMPINFO RANGESORT
2. ∀ALIST.LISTP RANGE(ALIST)

;labels: SIMPINFO APPALISTSORT
5. ∀ALIST Y.SEXP APPALIST(Y,ALIST)
```

proof APPL

```
;labels: SIMPINFO APPLFACTS
3. ∀U I.I<LENGTH U⊃SEXP APPL(U,I)∧MEMBER(APPL(U,I),U)
```

Simpinfo from file MULT

proof MULTIPLICITY

```
;labels: SIMPINFO MULTFACT
3. ∀U A.NATNUM(MULT(U,A))

;labels: SIMPINFO EMPTYFACTS
7. ∀U.MULT(U,EMPTYSET)=0
```

Simpinfo from file ASSOC

proof ALISTFACTS

```
;labels: SIMPINFO COMPALISTSORT
11. ∀ALIST ALIST1.ALISTP ALIST ∞ ALIST1
```

Simpinfo from file PERMF proof PERMFACTS

```
;labels: SIMPINFO SORTCOMP
2. ∀V U.DEF_APPL(V,U)⊃LISTP V•U

3. ∀V U.ALLP(λX.NATNUM(X)∧X<LENGTH V,U)⊃LISTP V•U

4. ∀M N.LISTP IDENT1(M,N)

5. ∀N.LISTP IDENT(N)

6. ∀U N I.LISTP INVERS1(U,I,N)

7. ∀U.LISTP INVERSE(U)

;labels: SIMPINFO IDENT_LENGTH
9. ∀N M.LENGTH (IDENT1(M,N))=N

10. ∀N.LENGTH (IDENT(N))=N
```

10.1. Index of Definitions.

Definitions from file LISPAX

proof LISPAX

```
;labels: LISTINDUCTIONDEF
34. ∀DF NILCASE DEF.
     (∃FUN.(∀PARS X U.FUN(NIL,PARS)=NILCASE(PARS)∧
                       FUN(X.U,PARS)=DEF(X,U,FUN(U,DF(X,PARS)),PARS)))
;labels: SEXPINDUCTIONDEF
36. ∀ATOMCASE DEFSEXP DF1 DF2.
     (∃FUN.
      (∀PARS X Y Z.(ATOM Z⊃ FUN(Z,PARS)=ATOMCASE(Z,PARS))∧
                   FUN(X.Y,PARS)=
                    DEFSEXP(X,Y,FUN(X,DF1(X,Y,PARS)),
                            FUN(Y,DF2(X,Y,PARS)),PARS)))
;labels: HIGH_ORDER_DEFINITION
40. ∀BIGFUN ATOM_FUN.
     (∃DEFINED_FUN.(∀X Y.(ATOM X⊃DEFINED_FUN(X)=ATOM_FUN(X))∧
                          DEFINED_FUN(X.Y)=
                           BIGFUN(X,Y,DEFINED_FUN(X),
                                  DEFINED_FUN(Y))))
;labels: SIMPINFO LISTDEF
45. ∀X LST.LIST(X,LST)=X.LIST(LST)

;labels: SIMPINFO APPENDEF
47. ∀X U V.NIL*V=V∧X.U*V=X.(U*V)

;labels: ALLPDEF
52. ∀PHI X U.ALLP(PHI,NIL)∧
             ALLP(PHI,X.U)=(IF PHI(X) THEN ALLP(PHI,U) ELSE FALSE)

;labels: SOMEPDEF
53. ∀PHI X U.¬SOMEP(PHI,NIL)∧
             SOMEP(PHI,X.U)=(IF PHI(X) THEN TRUE ELSE SOMEP(PHI,U))

;labels: MAPCARDEF
54. ∀FN X U.MAPCAR(FN,NIL)=NIL∧MAPCAR(FN,X.U)=FN(X).MAPCAR(FN,U)

;labels: ALISTDEF
57. ∀ALIST.¬NULL ALIST⊃
           ¬ATOM CAR ALIST∧ATOM CAR (CAR ALIST)∧ALISTP CDR ALIST

;labels: SIMPINFO ALISTDEF
58. ∀XA Y ALIST. ALISTP NIL ∧ALISTP (XA.Y).ALIST

;labels: ASSOCDEF
60. ∀X XA Y ALIST.ASSOC(X,NIL)=NIL∧
                  ASSOC(X,(XA.Y).ALIST)=
                   (IF X=XA THEN XA.Y ELSE ASSOC(X,ALIST))

;labels: MEMBERDEF
63. ∀X Y U.¬MEMBER(X,NIL)∧MEMBER(X,Y.U)=(X=Y∨MEMBER(X,U))

;labels: UNIQUENESSDEF
65. ∀U X.UNIQUENESS(NIL)∧(UNIQUENESS(X.U)≡¬MEMBER(X,U)∧UNIQUENESS(U))
```

Definitions from file SET

proof SETS

```
;labels: EPSILONDEF
6. ∀AV XV.XV∈AV≡AV(XV)

;labels: INTERDEF
9. ∀AV BV.AV∩BV=(λXV.AV(XV)∧BV(XV))

;labels: UNIONDEF
11. ∀AV BV.AV∪BV=(λXV.AV(XV)∨BV(XV))

;labels: INCLUSIONDEF
13. ∀AV BV.AV⊂BV≡(∀XV.AV(XV)⊃BV(XV))

;labels: EMPTYSETDEF
14. EMPTYSET=(λXV.FALSE)

;EMPTYP:
15. ∀AV.EMPTYP(AV)=(∀XV.¬AV(XV))

;labels: MKSET_DEF
17. ∀XV.MKSET(XV)=(λYV.YV=XV)

;labels: MKLSETDEF
19. ∀U.MKLSET(U)=(λX.MEMBER(X,U))
```

Definitions from file NATNUM

proof NATNUM

```
;labels: SIMPINFO PRED_DEF
19. ∀N.PRED(N')=N

;labels: SIMPINFO PLUSFACTS PLUSDEF
21. ∀N K.0+N=N∧K'+N=(K+N)'

;labels: SIMPINFO PLUSFACTS PLUSDEF1
24. ∀N.1+N=N'∧N+1=N'

;labels: TIMESFACTS
30. ∀N K.0*N=0∧N'*K=N*K+K
```

Definitions from file MINUS

proof LESSEQ

```
;labels: LESSEQDEF
3. ∀M N.M≤N=(M=N∨M<N)
```

proof MINUS

```
;labels: MINUSDEF
2. ∀M N.M-0=M∧M-N'=PRED(M-N)
```

Definitions from file LENGTH

proof LENGTH

```
;labels: SIMPINFO LENGTHDEF
2. ∀U X.LENGTH NIL=0∧LENGTH (X.U)=LENGTH U'
```

proof INDUCTION

```
;labels: INDUCTIVE_DEFINITION
5. ∀NDF ZCASE NDEF.
    (∃FUN.(∀NPARS N.FUN(0,NPARS)=ZCASE(NPARS)∧
                    FUN(N',NPARS)=
                    NDEF(N,FUN(N,NDF(N,NPARS)),NPARS)))

;labels: HIGH_ORDER_NATNUM_DEFINITION
10. ∀INDFN ARB.
     (∃DEF_FUN.(∀N.DEF_FUN(0)=ARB∧
                   DEF_FUN(N')=INDFN(N,DEF_FUN(N))))
```

Definitions from file NTH

proof NTH

```
;labels: SIMPINFO NTHDEF
2. ∀X U N.NTH(NIL,N)=NIL∧NTH(U,0)=CAR U∧NTH(X.U,N')=NTH(U,N)
```

proof NTHCDR

```
;labels: SIMPINFO NTHCDRDEF
2. ∀X U N.NTHCDR(NIL,N)=NIL∧NTHCDR(U,0)=U∧NTHCDR(X.U,N')=NTHCDR(U,N)
```

proof FSTPOSITION

```
;labels: FSTPOSITIONDEF
2. ∀X U Y.FSTPOSITION(NIL,Y)=NIL∧
          FSTPOSITION(X.U,Y)=
           (IF ¬MEMBER(Y,X.U) THEN NIL
               ELSE (IF X=Y THEN 0 ELSE FSTPOSITION(U,Y)'))
```

proof INJ

```
;labels: INJDEF
2. ∀U.INJ(U)=(∀N M.N<LENGTH U∧M<LENGTH U∧NTH(U,N)=NTH(U,M)⊃N=M)
 NIL)
```

Definitions from file APPL

proof APPALIST

```
;labels: DOMDEF
2. ∀XA Y ALIST.DOM(NIL)=NIL∧DOM((XA.Y).ALIST)=XA.DOM(ALIST)

;labels: RANGEDEF
4. ∀XA Y ALIST.RANGE(NIL)=NIL∧RANGE((XA.Y).ALIST)=Y.RANGE(ALIST)

;labels: FUNCTDEF
6. ∀ALIST.FUNCTP(ALIST)≡UNIQUENESS(DOM(ALIST))

;labels: INJECTDEF
8. ∀ALIST.INJECTP(ALIST)≡FUNCTP(ALIST)∧UNIQUENESS(RANGE(ALIST))

;labels: APPALISTDEF
10. ∀ALIST Y.APPALIST(Y,ALIST)=CDR ASSOC(Y,ALIST)

;labels: SAMEMAPDEF
12. ∀ALIST ALIST1.SAMEMAP(ALIST,ALIST1)≡
                 MKLSET(DOM(ALIST))=MKLSET(DOM(ALIST1))∧
                 (∀Y.Y∈MKLSET(DOM(ALIST))⊃
                  APPALIST(Y,ALIST)=APPALIST(Y,ALIST1))

;labels: PERMUTP_DEF
13. ∀ALIST.PERMUTP(ALIST)≡
          FUNCTP(ALIST)∧MKLSET(DOM(ALIST))=MKLSET(RANGE(ALIST))
```

proof ALISTFACTS

```
;labels: SAMEMAP_DEF1
10. ∀ALIST1 ALIST2.SAMEMAP(ALIST1,ALIST2)≡
                   MKLSET(DOM(ALIST1))=MKLSET(DOM(ALIST2))∧
                    (∀X.APPALIST(X,ALIST1)=APPALIST(X,ALIST2))
```

proof APPL

```
;labels: APPLDEF
1. ∀U I.APPL(U,I)=NTH(U,I)

;labels: INTODEF
5. ∀U.INTO(U)=(∀N.N<LENGTH U⊃NATNUM(NTH(U,N))∧NTH(U,N)<LENGTH U)

;labels: ONTODEF
7. ∀U.ONTO(U)=(INTO(U)∧(∀N.N<LENGTH U⊃MEMBER(N,U)))

;PERM
9. ∀U.PERM(U)=ONTO(U)
```

Definitions from file SUMS

proof SUMS

```
;labels: ALLNUMDEF
7. ∀N A.ALLNUM(0,A)∧(ALLNUM(N',A)≡A(N)∧ALLNUM(N,A))

;labels: SOMENUMDEF
8. ∀N A.¬SOMENUM(0,A)∧(SOMENUM(N',A)≡A(N)∨SOMENUM(N,A))

;labels: SUMDEF
9. ∀N NUMSEQ.SUM(NUMSEQ,0)=0∧
             SUM(NUMSEQ,N')=SUM(NUMSEQ,N)+NUMSEQ(N)

;labels: UNDEF
10. ∀N SETSEQ.UN(SETSEQ,0)=EMPTYSET∧
             UN(SETSEQ,N')=UN(SETSEQ,N)∪SETSEQ(N)

;labels: DIJPAIR_DEF
12. ∀A B.DISJ_PAIR(A,B)=EMPTYP(A∩B)

;labels: DISJOINTDEF
14. ∀N SETSEQ.DISJOINT(SETSEQ,0)∧
              DISJOINT(SETSEQ,N')=
               (DISJOINT(SETSEQ,N)∧DISJ_PAIR(UN(SETSEQ,N),SETSEQ(N)))
```

Definitions from file MULT

proof MULTIPLICITY

```
;labels: MULT_DEF
2. ∀X U A.MULT(NIL,A)=0∧
          MULT(X.U,A)=(IF A(X) THEN MULT(U,A)' ELSE MULT(U,A))
```

Definitions from file ASSOC

```
;labels: COMPALISTDEF
2. ∀ALIST1 ALIST2 XA Y.NIL ∞ ALIST2=NIL∧
                       ((XA.Y).ALIST1) ∞ ALIST2=
                        (XA.APPALIST(Y,ALIST2)).ALIST1 ∞ ALIST2

;labels:  INVALISTDEF
4. ∀ALIST XA Y.INVALIST(NIL)=NIL∧
               INVALIST((XA.Y).ALIST)=(Y.XA).INVALIST(ALIST)

;labels: IDALISTPDEF
6. ∀ALIST XA Y.IDALISTP(NIL)∧
               (IDALISTP((XA.Y).ALIST)≡XA=Y∧IDALISTP(ALIST))
```

Definitions from file PERMP

proof COMP PRED

```
;labels: COMPDEF
2. ∀U V W.COMP(U,V,W)≡LENGTH U=LENGTH W∧
                      (∀N.N<LENGTH U⊃NTH(U,N)=NTH(V,NTH(W,N)))

;labels: ID_DEF
4. ∀U.ID(U)≡(∀N.N<LENGTH U⊃NTH(U,N)=N)

;labels: INVDEF
6. ∀U V.INV(U,V)≡(∀N.N<LENGTH U⊃NTH(U,N)=FSTPOSITION(V,N))
```

Definitions from file PERMF

proof COMP FNCT

```
;labels: DEF_APPL_FACT
2. ∀U V.DEF_APPL(V,U)≡ALLP(λX.NATNUM(X)∧X<LENGTH V,U)

;labels: COMPOSEDEF
4. ∀U V X.U•NIL=NIL∧U•(X.V)=NTH(U,X).U•V

;labels: IDENTDEF1
6. ∀X U N I.IDENT1(I,0)=NIL∧IDENT1(I,N')=I.IDENT1(I',N)

;labels: IDENTDEF
8. ∀N.IDENT(N)=IDENT1(0,N)

;labels: INVERSDEF1
10. ∀U I N.INVERS1(U,I,0)=NIL∧INVERS1(NIL,I,N)=NIL∧
           INVERS1(U,I,N')=
           (IF NULL FSTPOSITION(U,I) THEN NIL
               ELSE FSTPOSITION(U,I).INVERS1(U,I',N))

;labels: INVERSDEF
12. ∀U.INVERSE(U)=INVERS1(U,0,LENGTH U)
```

INDEX OF FORMULAS

. FORMULA INDEX

FORMULA INDEX