Constructive Automata Theory Implemented with the Nuprl Proof Development System

Christoph Kreitz

TR 86-779
September 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

Constructive Automata Theory
implemented with the
Nuprl Proof Development System

Experience report

by

Christoph Kreitz

August 1986

Department of Computer Science
Cornell University
Ithaca NY 14853

Abstract

The Nuprl proof development system was designed for the computer-assisted problem solving in mathematics and programming. In particular it can be used for the development of mathematical proofs and of programs which are guaranteed to meet their specifications. The implementation of the theory of finite automata gave lots of insights into its strengths and weaknesses and shows that Nuprl is indeed powerful enough now to obtain nontrivial results within reasonable amounts of time. Its success shall encourage people to actually use the system and build theories within it.

This report describes the techniques and the user-defined extensions to the Nuprl object language which were necessary to formulate and prove theorems from the theory of finite automata. It also describes the experiences which came from actually working with the current Nuprl system and gave some useful insights into its strengths and weaknesses. A complete Nuprl-proof of the pumping lemma and its computational evaluation are presented and an outline for future development is given.

Literature quotes refer to:

[1] The PRL-Staff: *Implementing Mathematics with the Nuprl Proof Development System*, Prentice Hall, 1986.

[2] J.E.Hopcroft & J.D.Ullman: *Introduction to Automata Theory, Languages and Computation*, Adison Wesley, 1979.

1. An overview over the Nuprl proof development system

For those less familiar with the Nuprl proof development system this chapter will give a brief description of it which mainly is abstracted from the Nuprl "manual" [1]. It might be helpful to read chapters 1,2,6 and 8 of this book in order to be able to understand details presented in this report.

Nuprl is a computer system developed to provide assistance with the activities involved with the process of problem solving. With it one can create formulas, proofs and terms in a formal theory of mathematics and express concepts formally in definitions, theorems, theories, books and libraries. Moreover the formal theory is sensitive to the computational meaning of terms, assertions and proofs, and the computer system is able to carry out the corresponding actions. Thus Nuprl includes programming but in a broader sense it is a system for implementing mathematics.

The formal object language used in Nuprl (*Type Theory*) is based on the lambda calculus with an additional type discipline. Basic elements of the language are *types* and *members* of types. Everything else is expressed in terms of these. That is, properties of objects are described by the type whose member they are. Types are arranged in a cumulative hierarchy of universes **U1⊆U2⊆U3**.., i.e every type occurs in a universe, universes themselves are types and the elements of universes are types. Atomic Nuprl types are **int**, **atom** and **void** (a type without members). Type constructors are used to build more types from these ones. If **A** and **B** are types so is **A list** (list of elements of **A**), **A#B** (cartesian products), **A|B** (disjoint union) and **A->B** (the functions from **A** to **B**). In addition to these constructors there are dependent products **x:A#B** (resembling $\Sigma x \epsilon A.B(x)$), which expresses that the type of the second member **b** of a pair **<a,b>** can depend on the value of **a**, and dependent functions **x:A->B** ($\Pi x \epsilon A.B(x)$). Set types **{x:A|B}** and quotient types **x,y:A//B** allow Nuprl to express the notion of constructive sets and equivalence classes. Further type constructors came from the idea that <u>propositons can formally be expressed as types</u>. **a=b in A** expresses the proposition that **a** is equal to **b** in the type **A**. **a in A** is short for **a=a in A** and **a<b** was especially added for integer arithmetic. Finally recursive types and the type of partial functions were recently implemented.

The assertions one tries to prove in Nuprl have to be formulated as *judgements* of the form $x_1:T_1.,...,x_n:T_n$ >> S where x_i are variables and T_i and S are terms. $x_1:T_1.,...,x_n:T_n$ is called *hypothesis list* and S is called the *conclusion* (or *goal*). Upon completion of a proof the system automatically extracts a term which contains the computational meaning of the assertion (the *extract term*). This term is never displayed but it is accessible via a **term_of** operation.

Proofs of judgements are constructed top-down. That is, given a desired goal one tries to refine it into (easier) subgoals using *proof rules* until no subgoals are left. The typical form of a rule looks like this

```
H >> T by <rule-name>
    H1 >> T1
    .
    Hk >> Tk
```

If k=0 the rule has no subgoals, i.e it completes this part of the proof.

There are several categories of rules: *Introduction* rules (giving conditions under which an object may be judged to be a canonical member of a type. This includes *Formation* rules which are the same for type objects as members of universes), *Elimination* rules (how to use objects of a canonical type which are given in a hypothesis), *Equality* rules (including *Computation* rules which reduce noncanonical objects like `(\x.b(x))(a)` into their canonical form `b(a)`) and some miscellaneous rules not associated with a particular type (e.g. `arith`, a decision procedure for arithmetic reasoning). See the account given in chapter 8 of [1] for more details.

The computer system itself is oriented to the interactive creation of linguistic objects like *definitions*, *theorems*, *proofs* and *libraries* on a terminal screen. It provides a window system, which offers views of these objects, a text editor and a proof editor, a library module, a command language in order to create and handle objects, a metalanguage and a function evaluator (the Nuprl object language is functional).

The definition facility (DEF-object in the library) allows to define new notations in the form of templates which can be invoked when entering text. It provides the option of using a "private" object language on top of Nuprl. The proof editing facility supports the top-down construction of proofs. During a proof the user types the name of a rule into a rule-window and the system responds with a list of subgoals. Then the user selects a subgoal and continues the proof until it is completed.

The metalanguage (ML) finally allows to write programs which manipulate objects of the Nuprl object language. Most importantly one can write ML-functions which search for or transform proofs. Such automated proof techniques and theorem proving techniques (called *tactics*) are extremly helpful for writing proofs. They might be used to fill in the details and the boring steps of a proof automatically and leave only the important parts of it to the user or to automate more difficult patterns of proofs that occur frequently. There are many more uses of tactics. However, given undecidability of the rich object language of Nuprl, one cannot hope to fully automate the theorem proving process. Like definitions tactics are an extension of the Nuprl object language (its rule-system) on top of it defined by the user. Tactics are applied in proofs in a way similar to proof rules. The implementation of the metalanguage makes sure that all proofs produced by tactics are in fact valid Nuprl proofs. A tactic may fail but it never produces invalid proofs. This makes it impossible to write or use tactics that produce incorrect results.

The current Nuprl system contains some predefined elementary tactics but they are far from being enough. Writing tactics and adding them to the library is an essential part of building a theory within Nuprl. It is nearly impossible to prove any major theorems without them.

2. Preparations

In order to avoid unnecessarily complicated proofs the actual implementation of theorems from a mathematical theory requires a lot of preparations. Many of the types occuring in the theory of finite automata do not belong to the original object language of Nuprl. Therefore I had to deal with them first before I could start proving the theorems I was actually interested in. In fact this turned out to be the major part of the work so far.

In computer science textbooks on elementary automata theory one will find a definition of deterministic finite automata (*dfa*) like this :

> *We formally denote a finite automaton by a 5-tuple* (Q,S,d,q_0,F) *where* Q *is a finite set of states,* S *is a finite input alphabet,* $q_0 \in Q$ *is the initial state,* $F \subseteq Q$ *is the set of final states and* d *is the transition function mapping* $Q \times S$ *to* Q. ([2] p.17)

The corresponding Nuprl definition of the type of deterministic finite automata would be expressed by dependent products:

```
Q:STATES # S:ALPHABETS # ( Q # S -> Q ) # Q # P(Q)
```

For convenience I took the alphabet `S` out of this definition and replaced it by a general alphabet `SYMBOLS`. The actual alphabet is rather irrelevant for the properties of deterministic finite automata and it is easier to prove properties of strings in general without having to care about the underlying alphabet. Strings however and the behavior of deterministic finite automata on strings are the main interest of automata theory. All the other constructs used in the definition above had to remain. The necessity for the finite set of states, the transition function and the initial state is obvious and the powerset construct for the final states (instead of having just one final state) is needed in the proof of the equivalence between nondeterministic and deterministic finite automata. Therefore definitions and theorems had to be developed for

- **finite sets** including a version of the pigeonhole principle which is needed for the proof of the pumping-lemma
- **sets** in general, mainly the powerset construct
- **strings** all kinds of operations on strings and their properties including inductive definition of string functions which is necessary for the extension of the transition function to strings

In order to have more readable versions of theorems and their proofs it was also very useful to extend the Nuprl object language to **logic** notations and **tupling** (mainly quadruples and functions on products).

Experience from working with the PRL system showed that theorems should not be proven without an exhaustive use of tactics. Otherwise a proof tree grows incredibly large and none of the original proof ideas remains visible. For example the first version of the proof of the pumping lemma which did not use tactics except for wellformedness goals resulted in a proof tree of depth greater than 30 and branching degree up to 8 (which came from sequencing in new facts). The new version, presented in chapter 4 has only 13 nodes altogether and reflects the original proof much better.

Using tactics and tacticals allows to combine all the formal steps one can overlook at the current node into a single "logical" step. This might result in a seemingly complicated rule but the resulting proof will be much smaller and thus better to understand - if there was a little bit of planing before. Furthermore proofs might be changed more easily and a proof by analogy can be done by simply editing text. Just copy the rule (or even a collection of them which have to be combined by **THEN/THENL** then) into a text buffer, e.g. a dummy **DEF** object in the library, then change the corresponding symbols and copy this into the rule window of the new proof. The proof of the pigeonhole principle in chapter 3.5 is an example for this.

It also turned out that using a big collection of small special purpose tactics makes more sense than applying one very powerful general purpose tactic everywhere. The reason again is readability of the proofs. General purpose tactics like **immediate** (or **membership** in chapter 3.1) are somewhat unpredictable in their behavior. They make proof-planning nearly impossible. However because of their search strategies they are great in finishing proofs of wellformedness and membership subgoals which would result in many boring explicit steps otherwise. The library **automata** therefore includes a collection of general and special purpose tactics which are described in the following chapter.

The extensions of Nuprl to logic, tupling, sets, finite sets, strings and - of course - deterministic finite automata were written as if new types would have been added to the object language. That is, they consist not only of some definitions and theorems but also of a collection of ML-functions and tactics for these types which simulate constructors, destructors, predicates and rules (formation, introduction, elimination and computation) for this new type. Thus the actual definition which is uninteresting for further purposes is kept invisible. Since the implementation of these rules uses only already existing rules, tactics and theorems the consistency of these rules follows immediately from the consistency of the original rule system. I did not have to bother with that problem. The concept of *user-defined PRL-extensions* therefore is extremely helpful. It allows to define a new (or private) object language on top of the existing system and then to reason just within that language one is more familiar with. A user now does not have to deal with the original type theory anymore and the system can be used for theorem proving and program development even by people who are not very familiar with the theory behind it.

3. The extensions to the PRL type system

This chapter describes definitions written to extend the Nuprl object language and some highlights of the theorems, rules and tactics involved in this process. The definitions presented here may not always be the optimal or most general formalization of the objects used in automata theory. They were not intended to be. They are one way to proceed and show what one has to watch for while generating new types. Often a Nuprl proof that does not work as intended reveals a misunderstanding in a definition and causes a reconsideration of it. A formalization of a definition may express not all the properties one has in mind. This happened several times while building automata theory in Nuprl and it still might happen to some of the definitions chosen now - depending on future applications and how carefully they were formalized. However the strict modularization of definitions and the corresponding rules (tactics) will keep the amount of necessary changes rather small. Only tactics directly involved with the altered definitions have to be rewritten.

Detailed descriptions of the rules can be found in the appendix B under the names given in the headline.

3.1 General extensions: ML-functions and tactics (supply.ml / tactics.ml)

The general tactics library contains a lot of PRL-rules, which were simply converted into tactics such that they may be used as arguments of tacticals, and three more powerful general purpose tactics, which use heuristics to build up a proof tree. The former ones - including some variants of the computation rule - are quite easy to understand from the description given in the appendix. The latter ones were written after some experience with Nuprl proofs. Very often I had ended up with subgoals which followed from an instantiation of a hypothesis or a theorem already proven (i.e. the subgoal was of the form $B[a_1,..,a_n/x_1,...,x_n]$ where one of the hypotheses or a theorem was $\forall x_1{:}A_1..\forall x_n{:}A_n.B$) or with a membership subgoal of the form **t in T** which was to be proven by a boring series of introduction steps sometimes giving type dependencies or new identifiers. The tactics **HYPOTHESIS**, **THEOREM** and **membership** use strategies which were a result of studying patterns in my own behavior finding the applicable rules in the manual while sitting in front of the screen. They include ML-functions for creating new identifiers (**new**), matching of the proofs conclusion against some term (**match_subterms**) and guessing the type dependencies needed in a particular rule (**type_of**). There are some variants of the tactics **THEOREM** and **HYPOTHESIS** in which the user has to give some of the information usually to be found by the search strategies. They are helpful if just some new facts have to be introduced or if the startegies would fail (or some other special purposes). The tactic **membership** is open to include subtactics dealing with membership problems of additional types first, e.g. with strings which results in the tactic **Wmember** (chapter 3.6). See appendix B under **tactics.ml** for details.

Five PRL-definitions were made up for the most frequently used constructs in rules

```
(*<comment>*)               ==
-<t:tactic>                 ==  <t> THEN autotactic
&<t:tactic>                 ==  THENMEMBER <t> membertac
@<t:tactic>                 ==  <t> THEN membertac
THEOREM <name>              ==  thm `<name>`
```

A comment should have no influence at all but it helps to explain a major proof idea within a rule. The **THEOREM** definition mainly was written to get rid of the token quotes in the tactic **thm** (a variant of the tactic **THEOREM** *). **autotactic** is supposed to be a fast tactic finishing trivial proofs of subgoals while **membertac** (currently equal to **membership**) handles more complicated proofs. Both tactics are set previously but the are subject to changes, i.e. improvements.

3.2 Logic (logic.ml)

The logic definitions are very similar to the ones described in [1] chapter 3.6. However I used the new facility of special symbols which was added in the meantime and allows having quantifiers like ∀ and ∃ instead of the names **all** and **some**.

```
<p:prop> & <q:prop>                                   ==  ((<p>)#(<q>))

∀<x:var>:<t:type>. <p:prop>                           ==  (<x>:(<t>)->(<p>))
∀<x:var>,<y:var>:<t:type>. <p:prop>                   ==  ∀<x>:<t>. ∀<y>:<t>. <p>
∀<x:var>,<y:var>,<z:var>:<t:type>. <p:prop>           ==  ∀<x>:<t>. ∀<y>,<z>:<t>. <p>

∃<x:var>:<t:type>. <p:prop>                           ==  (<x>:(<t>)#(<p>))
∃<x:var>,<y:var>:<t:type>. <p:prop>                   ==  ∃<x>:<t>. ∃<y>:<t>. <p>
∃<x:var>,<y:var>,<z:var>:<t:type>. <p:prop>           ==  ∃<x>:<t>. ∃<y>,<z>:<t>. <p>

<p:prop> => <q:prop>                                  ==  ((<p>)->(<q>))

<p:prop> <=> <q:prop>                                 ==  <p> => <q> & <q> => <p>

false                                                 ==  void

¬<p:prop>                                             ==  ((<p>)->false)

TYPE                                                  ==  U1
```

Most of the rules for the logic extension are obvious and not worth any further discussion. Mainly I got rid of having to give new identifiers and superfluous hypotheses were thinned out - a technique which probably should generally be provided by the PRL-system. Only for the repeated introduction for existential quantifiers I had to write a more complicated rule in order to save a lot of unnecessary work in a proof:

A proof of the goal **H >> ∃x1:A1..∃xn:An.B** by giving witnesses $a_1,..,a_n$ step by step would result in the following subgoals

$$
\begin{array}{l}
\gg a_1 \text{ in } A_1 \\
\quad . \\
\quad \gg a_n \text{ in } A_n[a_1,..,a_{n-1}/x_1,..,x_{n-1}] \\
\quad \gg B[a_1,..,a_n/x_1,..\ x_n] \\
\quad x_n{:}A_n \gg B[a_1,..,a_{n-1}/x_1,..,x_{n-1}] \text{ in } Ui \\
\quad . \\
x_1{:}A_1 \gg (\exists x_2{:}A_2.\ .\exists x_n{:}A_n.B) \text{ in } Ui
\end{array}
$$

In order to prove the last n subgoals one would now have to perform a lot of introduction steps and then prove $B[a_1,..,a_j/x_1,..,x_j]$ **in Ui** for $1 \le j \le n-1$, $A_n[a_1,..,a_j/x_1,..,x_j]$ **in Ui** for $1 \le j \le n-2$ etc. By sequencing goals like $\forall x_1{:}A_1..\forall x_n{:}A_n.(B \text{ in } Ui)$ in advance this can be reduced to the necessary

*Unfortunately I chose the same name twice. The distinction can be found by looking for token quotes around the lemma name.

steps, i.e. just one variant of each of these subgoals has to be proven. The result is the rule **repeat_some_intro** as described below:

```
H >> ∃x1:A1..∃xn:An.B by   repeat_some_intro i [a1;..;an]
      >> A1 in Ui
      x1:A1 >> A2 in Ui
       .
      x1:A1,..,xn:An >> B in Ui
      >>a1 in A1
       .
      >>an in An[a1,..,an-1/x1,..,xn-1]
      >>B[a1,..,an/x1,..xn]
```

The actual implementation of this rule is rather difficult because many cases and subproofs have to be considered carefully. Applications of this valuable rule can be found in the proofs of the pigeonhole principle (chapter 3.5) and of the pumping lemma (chapter 4 - the variant **word_some_intro** is used).

There are some tactics for logic which were written mainly to improve the readability of proofs and to deal with membership problems seperately. The tactic(al) **Allintro**, for example, reduces the number of subgoals after introducing a universal quantifier to <u>one</u>.

```
H >> ∀x:A.B     by AllIntro j AUjtac
   x:A >> B
```

provided that **AUjtac** is able to prove **A in Uj** completely. **Allintro** therefore is the key tactic to straightforward introduction of all-quantifiers in theorems. There are variants for particular types (like **int** or **WORDS** in chapter 3.6), for types where **membership** can handle the wellformedness subgoal and repeated versions. Similar tactics which get rid of the PRL-secific subgoals exist for introduction of existence problems, implication and equivalence.

<u>3.3 Tupling</u> (tupling.ml)

Tupling deals with an extension of the pair- and spread-definition to triples, quadruples, functions on pairs as far as it is necessary for the automata theory library. It helps avoiding multiple brackets or spread expressions and also repetitious application of (nearly) the same rules in a proof.

```
(<a:term>,<b:term>)                              ==  ((<a>,<b>))
(<a:term>,<b:term>,<c:term>)                     ==  (<a>,(<b>,<c>))
(<a:term>,<b:term>,<c:term>,<d:term>)            ==  (<a>,(<b>,<c>,<d>))

<t:term> -where (<a:id>,<b:id>)=<x:term>-        ==  spread(<x>;<a>,<b>.<t>)
<t:term>- where (<a:id>,<b:id>,<c:id>)=<x:term>-
                                        ==  <t> -where (<b>,<c>)=tp2- -where (<a>,tp2)=<x>-
<t:term>- where (<a:id>,<b:id>,<c:id>,<d:id>)=<x:term>-
                                        ==  <t>- where (<b>,<c>,<d>)=tp3- -where (<a>,tp3)=<x>-

λ<x:id>,<y:id>.<t:term>                          ==  (\xx.<t> -where (<x>,<y>)=xx- )
```

Like the logic rules the rules for tupling are used mostly as supplement tactics in the other extensions of Nuprl. Mainly they use techniques combining hypotheses like $x = (a,x_1)$ in $y_1{:}A_1\#y_2{:}A_2\#y_3{:}A_3\#A_4$, $x_1 = (b,x_2)$ in $y_2{:}A_2[a/y_1]\#y_3{:}A_3[a/y_1]\#A_4[a/y_1]$ and $x_2 = (c,d)$ in $y_3{:}A_3[a,b/y_1,y_2]\#A_4[a,b/y_1,y_2]$ into the one hypothesis one is interested in, which is $x = (a,b,c,d)$ in $y_1{:}A_1\#y_2{:}A_2\#y_3{:}A_3\#A_4$, and computing "using"-types in intermediate steps. Some additional computation rules and tactics are special formulations of the general PRL-rules **compute** and **computehyp**.

The extension so far only consists of the definitions and tactics which were necessary for my purposes. It should be completed to a more powerful tool.

3.4 Sets (sets.ml)

The implementation of sets showed that for an exact formulation some distinctions have to be made which are normally ignored in computer science textbooks. For example the intersection of two sets often is simply expressed by $A \cap B = \{x \mid x \in A \,\&\, x \in B\}$. However in type theory (and also in order to avoid Russel's paradoxon) the type of x has to be specified. Therefore set-theory in PRL has to be formulated with respect to a "context"-type. Secondly the predicate x∈A cannot be substituted simply by **x in A** since this is only wellformed if x actually belongs to the set (type) A. This finally led to the definition of sets by a pair (carrier,pred) where carrier gives the context and pred is a predicate on carrier expressing the relation x∈A. In analogy to conventional set-notation the definition {x:carrier||p} stands for the pair (carrier,\x.p). A conversion from pairs into a set-type of PRL is necessary if access to the elements of a SET is required.

The definitions given below and the corresponding tactics include only the set theoretic notions necessary for the implementation of automata theory so far.

```
SETS                                    ==  (carrier:TYPE # (carrier -> U1))

{<x:id>:<carrier:TYPE>||<pred:TYPE>}    ==  (<carrier>,(\ <x>.<pred>))
{<el:element>:<carrier:TYPE>}           ==  {x_el:<carrier>||x_el = <el> in <carrier>}

carrier(<X:SETS>)                       ==  carrier -where (carrier,pred)=<X>-
pred(<X:SETS>)                          ==  pred -where (carrier,pred)=<X>-

<x:element> ε {<X:SETS>}                ==  pred(<X>)(<x>)
{<X:SETS into a set>}                   ==  {x:carrier|pred(x)} -where (carrier,pred)=<X>-

P(<T:TYPE>)                             ==  {S:SETS|carrier(S) = <T> in TYPE}
```

Note that generally the predicate of a SET is not decidable, i.e. **xεA|¬xεA** does not hold for x:carrier(A). If this property is required further conditions have to be given. For example one could define DECIDABLE SETS == {S:SETS| ∀x:carrier(S).S} and correspondingly decidable powersets as a subset of them. The SET-definitions also could easily be extended to other operations on **SETS** as follows:

```
AUB    == {x:carrier(A)| (xεA | xεB)}          (Union)
A∩B    == {x:carrier(A)| (xεA & xεB)}          (Intersection)
A^C    == {x:carrier(A)| ¬xεA }                (Complement)
A⊆B    ==  ∀x:carrier(A).xεA => xεB            (Subset)
P(A)   ==  {S:SETS| S⊆A}                       (alternate powerset definition)
```

where A and B are SETS assumed to have the same carrier.

The two versions to define powersets come from two different points of view. In the first one, which is used in the definition of finite automata, one is interested in sets having the same carrier which may be not a member of **SETS** while the other one allows studying set theoretic properties of powersets within the same carrier as before. It would be interesting to extend this to a full Nuprl set-theory library.

3.5 Finite sets (finset.ml)

Finite sets cause proplems similar to the ones of general set theory. Given a finite set Q one is interested in its cardinality - a proof for being finite - and in access to its elements. Since the Nuprl set-constructor **{x:A|B}** generally does not allow to use the information B in proofs, a definition of FINITE SETS as a subset of SETS does not seem to be reasonable. There would be no way to compute the cardinality of a finite set.

In order to avoid spending all my time in dealing with minor problems instead of implementing what I originally wanted to do (automata theory), I finally chose the easiest possible definition sufficient for my purposes. Often it is enough to know that a finite set Q has n distinct members while the specific elements are less interesting. Therefore a finite set is represented by its cardinality n which generally stands for the set {1,..,n}.

The rules corresponding to these definitions

```
<i:int>≤<j:int>                     ==  ((<i>-1) < (<j>)
<x1:int>≠<x2:int>                   ==  ¬(<x1>) = (<x2>) in int

N                                   ==  {nn:int | -1 < nn}
{1,...,<bound:N>}                   ==  {i:int | 0<i & i≤<bound> }
{0,..,<r:N>}                        ==  {num:int | -1 < num & num≤<r>}

FINITE SETS                         ==  N
{<A:FINITE SETS into a set>}        ==  {1,..,<A>}
```

The rules corresponding to these definitions are not written yet. There is however a collection of theorems dealing with properties of finite sets which later might be used in specific computation rules. Mostly they express properties which are intuitively clear but of course unknown to the Nuprl system, for example the following ones:

* **fineq**
  `THM >>∀m:N. ∀x:{1,..,(m-1)+1}. x in {1,..,m}`
* **extend**
  `THM >>∀k:N. ∀y:{1,..,k}. y in {1,..,k+1}`
* **restrict**
  `THM >>∀m:N. ∀x:{1,..,m}. x<m => x in {1,..,m-1}`
* **restrict1**
  `THM >>∀m:N. ∀x:{1,..,m}. x≠m => x in {1,..,m-1}`
* **finarith**
  `THM >>∀m:N. ∀x:{1,..,m}. ( x = m in {1,..,m}) | (x<m)`
* **hole1**
  ```
  THM >>∀k:int.0≤k =>  ∀n:N. ∀f:{1,..,k}->{1,..,n}. ∀y:{1,..,n}
        . ∃x:{1,..,k}. f(x)=y in int |  ∀x:{1,..,k}. f(x)≠y
  ```

Some of them deal with problems of integer arithmetic the **arith**-rule should be able to handle.

* **int_arith**
  `THM >>∀x,y:int. (x = y in int) | x≠y`
* **intsub**
  `THM >>∀i,j:int. i=j in int => j-i = 0 in int`
* **lessarith**
  `THM >>∀i,j:int. i<j => 0<j-i`
* **lessarith1**
  `THM >>∀i,j,k:int. i≤j & 0≤k => i≤j+k`

Appendix A, chapters 1.2.b-1.2.c give a complete list of all the theorems.

Probably the most interesting theorem is a basic version of the pigeonhole principle which is essential for the proof of the pumping lemma.

* **pigeon**
  `THM >>∀k:int. 0≤k => ∀f:{1,..,k+1} -> {1,..,k}. ∃i,j:{1,..,k+1}. i<j & f(i) = f(j) in int`

A proof of this theorem which was built up using only primitive inference rules is sketched in chapter 11 of [1]. The proof presented here demonstrates some of the advantages of exhaustively using tactics over this method. It is much shorter and since parts of it are symmetric but not worth formulating a lemma the method of analogy proofs by just copying and editing text could be applied. It also reflects the original proof much better since PRL-specific technical details like wellformedness problems could successfully be suppressed by tactics.

The proof begins with an induction on `k` and the introduction of the function `f`. The implicit tactic **membership** shows that the down case `(k<0)` contradicts the assumption `0≤k`.

```
+-------------------------------------------------------------------------+
|EDIT THM pigeon                                                          |
|-------------------------------------------------------------------------|
|* top                                                                    |
|>> ∀k:int.0≤k => ∀f:{1,..,k+1}->{1,..,k}. ∃i,j:{1,..,k+1}. i<j & f(i)=f(j) in int|
|                                                                         |
|BY        int_all_intro                                                  |
|   THEN  @Elim 1                                                         |
|   THENL [ -Seq [FALSE]; Elim 3 THENL [membership; Thinning [3;4]] ]     |
|                                                                         |
|1* 1. k:int                                                              |
|   2. 0≤0                                                                |
|   3. f:{1,..,0+1} -> {1,..,0}                                           |
|   >> false                                                              |
|                                                                         |
|2* 1. k:int                                                              |
|   2. 0<k                                                                |
|   3. f:{1,..,k+1} -> {1,..,k}                                           |
|   4. ∀f:{1,..,(k-1)+1}->{1,..,k-1}. ∃i,j:{1,..,(k-1)+1}. i<j & f(i)=f(j) in int|
|   >> ∃i,j:{1,..,k+1}. i<j & f(i) = f(j) in int                          |
|                                                                         |
+-------------------------------------------------------------------------+
```

The base case `(k=0)` is easy because there is no function `f:{1,..,0+1} -> {1,..,0}`.

```
+-------------------------------------------------------------------------+
|EDIT THM pigeon                                                          |
|-------------------------------------------------------------------------|
|* top 1                                                                  |
|1. k:int                                                                 |
|2. 0≤0                                                                   |
|3. f:{1,..,0+1} -> {1,..,0}                                              |
|>> false                                                                 |
|                                                                         |
|BY @Hyp_elim_on ['1'] 3 THEN @(Elim 3 THENL [Cumulativity; and_elim 5])  |
|                                                                         |
+-------------------------------------------------------------------------+
```

The up case is the only interesting one. There are two cases to consider:
`f(x)=f(k+1)` for some `x in {1,..,k}` or `f(x)≠f(k+1)` for all `x in {1,..,k}`

```
+---------------------------------------------------------------------------+
|EDIT THM pigeon                                                            |
|---------------------------------------------------------------------------|
|* top 2                                                                    |
|1. k:int                                                                   |
|2. 0<k                                                                     |
|3. f:{1,..,k+1} -> {1,..,k}                                                |
|4. ∀f:{1,..,(k-1)+1}->{1,..,k-1}. ∃i,j:{1,..,(k-1)+1}. i<j & f(i)=f(j) in int|
|>> ∃i,j:{1,..,k+1}. i<j & f(i) = f(j) in int                               |
|                                                                           |
|BY Cases ['∃x:{1,..,k}. f(x)=f(k+1) in int';'∀x:{1,..,k}. f(x)≠f(k+1)' ]   |
|                                                                           |
|1* >> (∃x:{1,..,k}. f(x)=f(k+1) in int) | (∀x:{1,..,k}. f(x)≠f(k+1))       |
|                                                                           |
|2* 5. ∃x:{1,..,k}. f(x)=f(k+1) in int                                      |
|   >> ∃i,j:{1,..,k+1}. i<j & f(i) = f(j) in int                            |
|                                                                           |
|3* 5. ∀x:{1,..,k}. f(x)≠f(k+1)                                             |
|   >> ∃i,j:{1,..,k+1}. i<j & f(i) = f(j) in int                            |
|                                                                           |
+---------------------------------------------------------------------------+
```

The proof of this is an application of the lemma **hole1** using the tactic **theorem**. In order to make the uninteresting subgoals disappear all the rules necessary are put together into one.

```
+---------------------------------------------------------------------------+
|EDIT THM pigeon                                                            |
|---------------------------------------------------------------------------|
|* top 2 1                                                                  |
|1. k:int                                                                   |
|2. 0<k                                                                     |
|3. f:{1,..,k+1} -> {1,..,k}                                                |
|4. ∀f:{1,..,(k-1)+1}->{1,..,k-1}. ∃i,j:{1,..,(k-1)+1}. i<j & f(i)=f(j) in int|
|>> (∃x:{1,..,k}. f(x)=f(k+1) in int) | (∀x:{1,..,k}. f(x)≠f(k+1))          |
|                                                                           |
|BY theorem `hole1` ['k';imp;'k';'(\x.f(x))';'f(k+1)']                      |
|   THENL [ Elim 5 THENL [@Cumulativity;&and_elim 7]                        |
|         ; Elim 5 THENL [@Cumulativity;&and_elim 7]                        |
|         ; -NormalizeHyp 5                                                 |
|         ]                                                                 |
|                                                                           |
+---------------------------------------------------------------------------+
```

The first of these two cases is trivial. `i=x` (from hypothesis 5 after elimination) and `j=k+1` have to be chosen. This is done by applying the tactic **repeat_some_intro** (see chapter 3.2). The subgoals mainly are problems of membership or finite sets. The general wellformedness of the goal-expression follows from a lemma **pigeon_type** which was formulated separately.

```
* pigeon_type
  THM >>∀k:N. ∀f:({1,..,k+1} -> {1,..,k}). ∀i,j:{1,..,k+1}. (i<j & f(i) = f(j) in int) in U1
```

```
+---------------------------------------------------------------------------+
|EDIT THM pigeon                                                            |
|---------------------------------------------------------------------------|
|* top 2 2                                                                  |
|1. k:int                                                                   |
|2. 0<k                                                                     |
|3. f:{1,..,k+1} -> {1,..,k}                                                |
|4. ∀f:{1,..,(k-1)+1}->{1,..,k-1}. ∃i,j:{1,..,(k-1)+1}. i<j & f(i)=f(j) in int|
|5. ∃x:{1,..,k}. f(x) = f(k+1) in int                                       |
|>> ∃i,j:{1,..,k+1}. i<j & f(i) = f(j) in int                               |
|                                                                           |
|BY (* choose i = x from hyp 5 and j = k+1  *)                              |
|                                                                           |
|         some_elim 5                                                       |
|   THEN  repeat_some_intro 1 ['x';'k+1']                                   |
|   THENL [ membership       ; membership                                   |
|         ; theorem `pigeon_type` ['k';'f';'i';'j']                         |
|         ; THEOREM extend  ; membership                                    |
|         ; -and_intro THEN theorem `wf2` ['k';'x']                         |
|         ]                                                                 |
|                                                                           |
+---------------------------------------------------------------------------+
```

In the other case the induction hypothesis has to be used. It requires a function `g:{1,..,k}->{1,..,k-1}`. `f` itself will possibly map some `x` in `{1,..,k}` to `k`. Then however, because of the assumption in hypothesis 5, `f(k+1)` will be in `{1,..,k-1}`. Thus `g(x):=(f(k+1) if f(x)=k,f(k+1) otherwise)` will do. The proof of this informal description requires some juggling with finite-set-arithmetic as performed in `top 2 3 1`. In preparation for the next step (`top 2 3 2`) also the induction hypothesis 4 is unraveled into the hypotheses 5-8.

```
+--------------------------------------------------------------------------+
|EDIT THM pigeon                                                           |
|--------------------------------------------------------------------------|
|* top 2 3                                                                 |
|1. k:int                                                                  |
|2. 0<k                                                                    |
|3. f:{1,..,k+1} -> {1,..,k}                                               |
|4. ∀f:{1,..,(k-1)+1} -> {1,..,k-1}. ∃i,j:{1,..,(k-1)+1}                   |
|   . i<j & f(i) = f(j) in int                                             |
|5. ∀x:{1,..,k}. f(x)≠f(k+1)                                               |
|>> ∃i,j:{1,..,k+1}. i<j & f(i) = f(j) in int                              |
|                                                                          |
|BY all_elim_on ['\x.int_eq(f(x);k;f(k+1);f(x))'] 4                        |
|   THENL                                                                  |
|   [ Intro THENL [IDTAC;membership]                                       |
|   ; some_elim 5 THEN and_elim 7 THEN NormalizeHyp 8                      |
|   ]                                                                      |
|                                                                          |
|1* 5. x:{1,..,(k-1)+1}                                                    |
|   >> int_eq(f(x);k;f(k+1);f(x)) in {1,..,k-1}                            |
|                                                                          |
|2* 5. i:{1,..,(k-1)+1}                                                    |
|   6. j:{1,..,(k-1)+1}                                                    |
|   7. i<j                                                                 |
|   8. int_eq(f(i);k;f(k+1);f(i)) = int_eq(f(j);k;f(k+1);f(j)) in int      |
|   >> ∃i,j:{1,..,k+1}. i<j & f(i) = f(j) in int                           |
|                                                                          |
+--------------------------------------------------------------------------+
```

```
+--------------------------------------------------------------------------+
|EDIT THM pigeon                                                           |
|--------------------------------------------------------------------------|
|* top 2 3 1                                                               |
|1. k:int                                                                  |
|2. 0<k                                                                    |
|3. f:{1,..,k+1} -> {1,..,k}                                               |
|4. ∀x:{1,..,k}. f(x)≠f(k+1)                                               |
|5. x:{1,..,(k-1)+1}                                                       |
|>> int_eq(f(x);k;f(k+1);f(x)) in {1,..,k-1}                               |
|                                                                          |
|BY Seq   ['f(x) in {1,..,k}']                                             |
|   THENL                                                                  |
|   [ theorem `extend1` ['k';'x']                                          |
|   ; Hyp_elim_on ['x'] 4                                                  |
|     THENL                                                                |
|     [ THEOREM fineq                                                      |
|     ; refine (int_eq_equality)                                           |
|       THENL                                                              |
|       [ theorem `wf1` ['k';'f(x)']                                       |
|       ; Equal                                                            |
|       ; THEOREM `restrict1`                                              |
|         THENL                                                            |
|         [membership; membership; theorem `finfuntype` ['k';'f';'k+1']]   |
|       ; THEOREM restrict1                                                |
|   ] ] ]                                                                  |
|                                                                          |
+--------------------------------------------------------------------------+
```

Now again a case analysis has to be done:

f(i)=k&f(j)=k or f(i)=k&f(j)≠k or f(i)≠k&f(j)=k or f(i)≠k&f(j)≠k

The validity of theses cases again is mainly a problem of finite-set-arithmetic. Trivial facts like f(i) in int have to be sequenced first which makes the rule look more complicated than it actually is.

```
+------------------------------------------------------------------------+
|EDIT THM pigeon                                                         |
|------------------------------------------------------------------------|
|* top 2 3 2                                                             |
|1. k:int                                                                |
|2. 0<k                                                                  |
|3. f:{1,..,k+1} -> {1,..,k}                                             |
|4. ∀x:{1,..,k}. f(x)≠f(k+1)                                             |
|5. i:{1,..,(k-1)+1}                                                     |
|6. j:{1,..,(k-1)+1}                                                     |
|7. i<j                                                                  |
|8. int_eq(f(i);k;f(k+1);f(i)) = int_eq(f(j);k;f(k+1);f(j)) in int       |
|>> ∃i,j:{1,..,k+1} . i<j & f(i) = f(j) in int                           |
|                                                                        |
|BY (* Consider four cases                                               |
|     1.f(i)=k&f(j)=k  2.f(i)=k&f(j)≠k  3.f(i)≠k&f(j)=k  4.f(i)≠k&f(j)≠k |
|     *)                                                                 |
|                                                                        |
|   THEOREM1 `finfuntype` ['k';'f']                                      |
|   THENL                                                                |
|   [ membership; Equal                                                  |
|   ; ElimOn 9 'i'                                                       |
|     THENL                                                              |
|     [ THEOREM extend1                                                  |
|     ; ElimOn 9 'j'                                                     |
|       THENL                                                            |
|       [ THEOREM extend1                                                |
|       ; Cases ['(f(i) = k in int)';'f(i)≠k']                           |
|         THENL                                                          |
|         [ THEOREM int_arith                                            |
|         ; Cases ['(f(j) = k in int)';'f(j)≠k']                         |
|           THENL [THEOREM int_arith;Thinning [10;11];Thinning [10]]     |
|         ; Cases ['(f(j) = k in int)';'f(j)≠k']                         |
|           THENL [THEOREM int_arith;Thinning [11];IDTAC]                |
|   ] ] ] ]                                                              |
|                                                                        |
|1* 9. ∀x:{1,..,k+1}. f(x) in int                                        |
|   10. f(i) = k in int                                                  |
|   11. f(j) = k in int                                                  |
|   >> ∃i,j:{1,..,k+1} . i<j & f(i) = f(j) in int                        |
|                                                                        |
|2* 9. ∀x:{1,..,k+1}. f(x) in int                                        |
|   10. f(j) in int                                                      |
|   11. f(i) = k in int                                                  |
|   12. f(j)≠k                                                           |
|   >> ∃i,j:{1,..,k+1}. i<j & f(i) = f(j) in int                         |
|                                                                        |
|3* 9. ∀x:{1,..,k+1}. f(x) in int                                        |
|   10. f(i) in int                                                      |
|   11. f(i)≠k                                                           |
|   12. f(j) = k in int                                                  |
|   >> ∃i,j:{1,..,k+1}. i<j & f(i) = f(j) in int                         |
|                                                                        |
|4* 9. ∀x:{1,..,k+1}. f(x) in int                                        |
|   10. f(i) in int                                                      |
|   11. f(j) in int                                                      |
|   12. f(i)≠k                                                           |
|   13. f(j)≠k                                                           |
|   >> ∃i,j:{1,..,k+1}. i<j & f(i) = f(j) in int                         |
|                                                                        |
+------------------------------------------------------------------------+
```

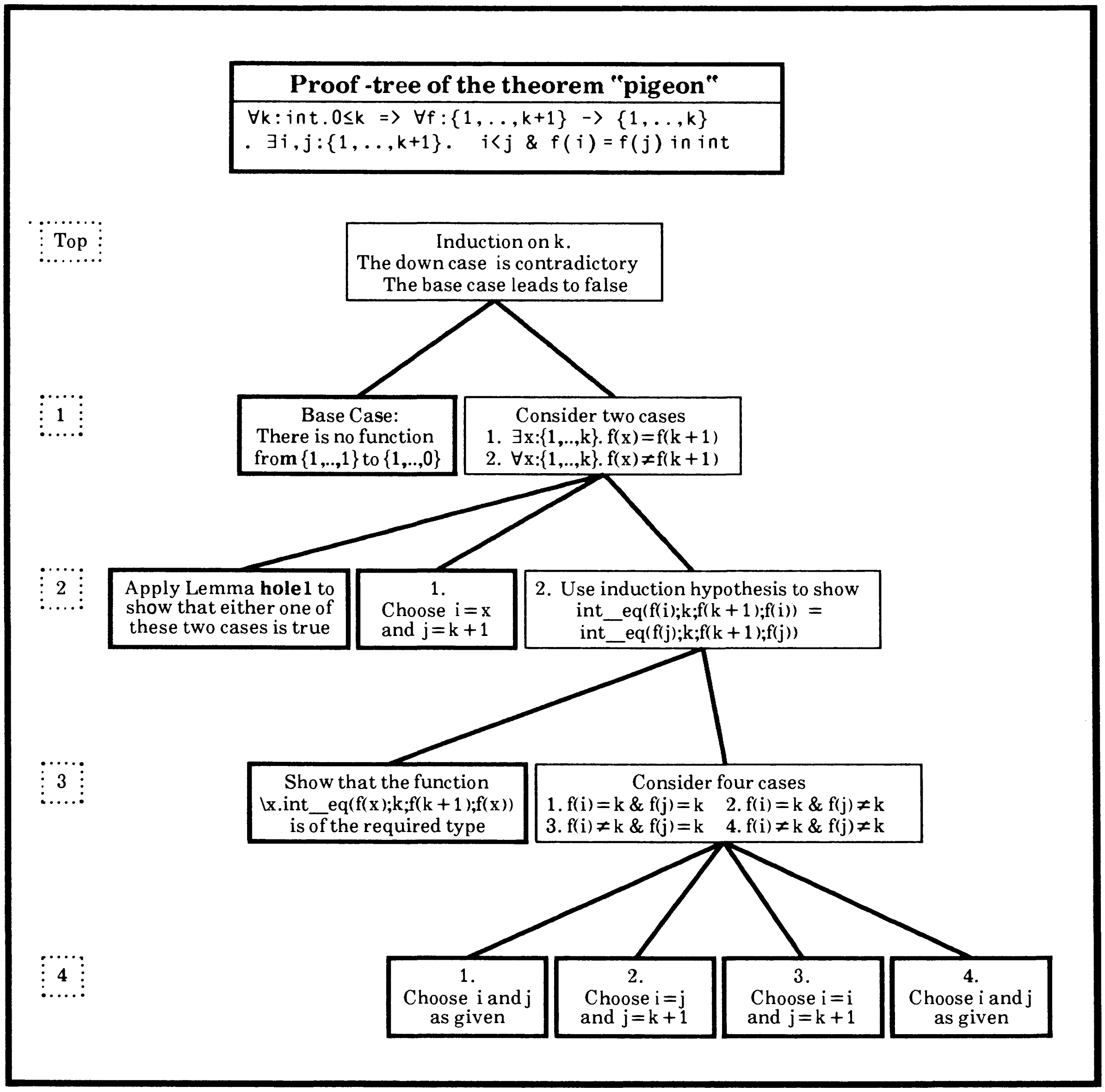
Now the rest of the proof is straightforward. For each case just the right values of i and j have to be chosen and `f(i)=f(j)` has to be shown by computing the `int_eq`-expression in hypothesis 8. The rules for each case were constructed using the "copy and edit text" method and the rule of `top 2 3 1`. Cases 2 and 3 are strongly symmetric and therefore this method is very effective here.

```
+---------------------------------------------------------------------------+
|EDIT THM pigeon                                                            |
|---------------------------------------------------------------------------|
|* top 2 3 2 1                                                              |
|1. k:int                                                                   |
|2. 0<k                                                                     |
|3. f:{1,..,k+1} -> {1,..,k}                                                |
|4. ∀x:{1,..,k}. f(x)≠f(k+1)                                                |
|5. i:{1,..,(k-1)+1}                                                        |
|6. j:{1,..,(k-1)+1}                                                        |
|7. i<j                                                                     |
|8. int_eq(f(i);k;f(k+1);f(i)) = int_eq(f(j);k;f(k+1);f(j)) in int          |
|9. ∀x:{1,..,k+1}. f(x) in int                                              |
|10. f(i) = k in int                                                        |
|11. f(j) = k in int                                                        |
|>> ∃i,j:{1,..,k+1} . i<j & f(i) = f(j) in int                              |
|                                                                           |
|BY (* choose i,j as given    *)                                            |
|                                                                           |
|   repeat_some_intro 1 ['i';'j']                                           |
|   THENL                                                                   |
|   [ membership      ; membership                                          |
|   ; theorem `pigeon_type` ['k';'f';'i';'j']                               |
|   ; THEOREM extend1 ; THEOREM extend1                                     |
|   ; -and_intro                                                            |
|   ]                                                                       |
|                                                                           |
+---------------------------------------------------------------------------+
```

```
+---------------------------------------------------------------------------+
|EDIT THM pigeon                                                            |
|---------------------------------------------------------------------------|
|* top 2 3 2 2                                                              |
|1. k:int                                                                   |
|2. 0<k                                                                     |
|3. f:{1,..,k+1} -> {1,..,k}                                                |
|4. ∀x:{1,..,k}. f(x)≠f(k+1)                                                |
|5. i:{1,..,(k-1)+1}                                                        |
|6. j:{1,..,(k-1)+1}                                                        |
|7. i<j                                                                     |
|8. int_eq(f(i);k;f(k+1);f(i)) = int_eq(f(j);k;f(k+1);f(j)) in int          |
|9. ∀x:{1,..,k+1}. f(x) in int                                              |
|10. f(j) in int                                                            |
|11. f(i) = k in int                                                        |
|12. f(j)≠k                                                                 |
|>> ∃i,j:{1,..,k+1}. i<j & f(i) = f(j) in int                               |
|                                                                           |
|BY (*    choose i=j and j=k+1    *)                                        |
|                                                                           |
|   repeat_some_intro 1 ['j';'k+1']                                         |
|   THENL                                                                   |
|   [ membership       ; membership                                         |
|   ; theorem `pigeon_type` ['k';'f';'i';'j']                               |
|   ; THEOREM extend1  ; membership                                         |
|   ; -and_intro                                                            |
|     THENL                                                                 |
|     [ Elim 6 THENL [@Cumulativity; &and_elim 14]          (* j<k+1 *)     |
|     ; Seq   ['int_eq(f(i);k;f(k+1);f(i))' = f(k+1) in int'                |
|             ;'int_eq(f(j);k;f(k+1);f(j)) = f(j) in int'  ]                |
|       THENL                                                               |
|       [ @Hyp_elim_on ['k+1'] 9 THEN -Inteq_computation 1 true             |
|       ; -Inteq_computation 1 false                                        |
|       ; Equal                                                             |
|   ] ] ]                                                                   |
|                                                                           |
+---------------------------------------------------------------------------+
```

```
+---------------------------------------------------------------------------+
|EDIT THM pigeon                                                            |
|---------------------------------------------------------------------------|
|* top 2 3 2 3                                                              |
|1. k:int                                                                   |
|2. 0<k                                                                     |
|3. f:{1,..,k+1} -> {1,..,k}                                                |
|4. ∀x:{1,..,k}. f(x)≠f(k+1)                                                |
|5. i:{1,..,(k-1)+1}                                                        |
|6. j:{1,..,(k-1)+1}                                                        |
|7. i<j                                                                     |
|8. int_eq(f(i);k;f(k+1);f(i)) = int_eq(f(j);k;f(k+1);f(j)) in int          |
|9. ∀x:{1,..,k+1}. f(x) in int                                              |
|10. f(i) in int                                                            |
|11. f(i)≠k                                                                 |
|12. f(j) = k in int                                                        |
|>> ∃i,j:{1,..,k+1}. i<j & f(i) = f(j) in int                               |
|                                                                           |
|BY (* choose i=i and j=k+1  *)                                             |
|                                                                           |
|   repeat_some_intro 1 ['i';'k+1']                                         |
|   THENL                                                                   |
|   [ membership        ; membership                                        |
|   ; theorem `pigeon_type` ['k';'f';'i';'j']                               |
|   ; THEOREM extend1   ; membership                                        |
|   ; -and_intro                                                            |
|     THENL                                                                 |
|     [ Elim 5 THENL [@Cumulativity; &and_elim 14]           (* i<k+1 *)    |
|     ; Seq   ['int_eq(f(i);k;f(k+1);f(i)) = f(k+1) in int'                 |
|             ;'int_eq(f(j);k;f(k+1);f(j)) = f(i) in int'  ]                |
|       THENL                                                               |
|       [ @Hyp_elim_on ['k+1'] 9 THEN -Inteq_computation 1 true             |
|       ; -Inteq_computation 1 false                                        |
|       ; Equal                                                             |
|   ] ] ]                                                                   |
|                                                                           |
+---------------------------------------------------------------------------+
```

```
+---------------------------------------------------------------------------+
|EDIT THM pigeon                                                            |
|---------------------------------------------------------------------------|
|* top 2 3 2 4                                                              |
|1. k:int                                                                   |
|2. 0<k                                                                     |
|3. f:{1,..,k+1} -> {1,..,k}                                                |
|4. ∀x:{1,..,k}. f(x)≠f(k+1)                                                |
|5. i:{1,..,(k-1)+1}                                                        |
|6. j:{1,..,(k-1)+1}                                                        |
|7. i<j                                                                     |
|8. int_eq(f(i);k;f(k+1);f(i))=int_eq(f(j);k;f(k+1);f(j)) in int            |
|9. ∀x:{1,..,k+1}. f(x) in int                                              |
|10. f(i) in int                                                            |
|11. f(j) in int                                                            |
|12. f(i)≠k                                                                 |
|13. f(j)≠k                                                                 |
|>> ∃i,j:{1,..,k+1}. i<j & f(i) = f(j) in int                               |
|                                                                           |
|BY (* choose i and j as given *)                                           |
|                                                                           |
|   repeat_some_intro 1 ['i';'j']                                           |
|   THENL                                                                   |
|   [ membership;      membership                                           |
|   ; theorem `pigeon_type` ['k';'f';'i';'j']                               |
|   ; THEOREM extend1; THEOREM extend1                                      |
|   ; -and_intro                                                            |
|     THEN                                                                  |
|     Seq   [ 'int_eq(f(i);k;f(k+1);f(i))= f(i) in int'                     |
|            ; 'int_eq(f(j);k;f(k+1);f(j))= f(j) in int' ]                  |
|     THENL [-Inteq_computation 1 false; -Inteq_computation 1 false; Equal] |
|   ]                                                                       |
|                                                                           |
+---------------------------------------------------------------------------+
```



Proof -tree of the theorem "pigeon"
∀k:int.0≤k => ∀f:{1,..,k+1} -> {1,..,k}
. ∃i,j:{1,..,k+1}. i<j & f(i)=f(j) in int
Top
Induction on k.
The down case is contradictory
The base case leads to false
1
Base Case:
There is no function
from {1,..,1} to {1,..,0}
Consider two cases
1. ∃x:{1,..,k}. f(x)=f(k+1)
2. ∀x:{1,..,k}. f(x)≠f(k+1)
2
Apply Lemma hole1 to
show that either one of
these two cases is true
1.
Choose i=x
and j=k+1
2. Use induction hypothesis to show
int__eq(f(i);k;f(k+1);f(i)) =
int__eq(f(j);k;f(k+1);f(j))
3
Show that the function
\x.int__eq(f(x);k;f(k+1);f(x))
is of the required type
Consider four cases
1. f(i)=k & f(j)=k 2. f(i)=k & f(j)≠k
3. f(i)≠k & f(j)=k 4. f(i)≠k & f(j)≠k
4
1.
Choose i and j
as given
2.
Choose i=j
and j=k+1
3.
Choose i=i
and j=k+1
4.
Choose i and j
as given

3.6 Strings (words_term.ml/ words_rules_1.ml/ words_rules_compute.ml/ words_tactics.ml/ words_recursion.ml)

The extension of the object language to strings is the most complete collection of definitions, theorems, "inference rules" and tactics in the automata theory library. It deals with **WORDS** - which is just another name for strings - and all kind of operations on them. I chose the type `int` to represent the standard alphabet **SYMBOLS**, mainly because it is a basic Nuprl type and allows an easy specification of strings. However, because the definitions and theorems dealing with strings are independent of any property of **SYMBOLS** (except that **SYMBOLS** has to be in **U1**) they work on any other type as well*. The fact that `int` is not a finite type is rather unimportant for string theory itself. It is easy to add conditions which guarantee that the symbols used in a theorem belong to a finite type if this property is really necessary.

Most of the definitions given below are self-explanatory. **ε** is the empty word; **⊗a** is the word solely consisting of the symbol a; **u∘v** is the concatenation of u and v; **w*a** means inserting the symbol a at the end of the word w (cons at the tail of w); **←w** is the word w in reverse order; **w↑i** is the i-fold iteration of w; **hd(w)** is the first symbol of w; **tl(w)** is the rest; **|w|** is the length of w; **w[i+1..lg]** is the substring of w resulting from cutting off the first i symbols; **w(i)** is the i-th symbol in w; **w[1..i]** is the substring of w consisting of the first i symbols; **w[l+1..r]** is the word w(l+1)...w(r).

```
SYMBOLS                          ==  int
WORDS                            ==  SYMBOLS list

ε                                ==  nil
<word:WORDS>≠ε                   ==  ∃a:SYMBOLS. ∃l:WORDS. <word> = a.l in WORDS

⊗<sym:symbol>                    ==  (<sym>.ε)
<lw:WORDS>∘<rw:WORDS>            ==  list_ind( <lw>;<rw>;hdl,tll,tll_rw.(hdl.tll_rw))
<word:WORDS>*<sym:SYMBOLS>       ==  (<word>∘⊗<sym>)
←<word:WORDS>                    ==  list_ind(<word>;ε;hdword,tlword,tlrevword. tlrevword*hdword)

<word:WORDS>↑<ii:N>              ==  ind(<ii>;k1,indhyp.ε;ε;k1,indhyp.(<word>∘indhyp))

hd(<word:WORDS>)                 ==  list_ind(<word>;0;hdword,tlword,hyp.hdword)
tl(<word:WORDS>)                 ==  list_ind(<word>;ε;hdword,tlword,hyp.tlword)
|<word:WORDS)>|                  ==  list_ind(<word>;0;hdword,tlword,lgtlword.(lgtlword+1))

<word:WORDS>[<ii:N>+1..lg]       ==  ind(<ii>;k1,w_i_lg.w_i_lg;<word>;k1,w_i_lg.tl(w_i_lg))
<word:WORDS>(<ii:N>)             ==  hd(<word>[<ii>-1+1..lg])
<word:WORDS>[1..<ii:N>]          ==  ind(<ii>;k1,w_1_k1_1.ε;ε;k1,w_1_k1_1.w_1__k1_1*<word>(k1))
<word:WORDS>[<lb:N>+1..<rb:N>]  ==  <word>[1..<rb><lb>+1..lg]

TRK(<g:A-\>B>,<h:B#SYMBOLS-\>B>)
                                 == (λx,w.list_ind(←w;<g>(x);endw,tlwrev,fxw.(<h>(fxw,endw))))
<h1:B#SYMBOLS-\>B>*              == TRK((\x.x),<h1>)
```

The last two definitions describe recursive definitions of functions on words in analogy to primitive recursion on natural numbers. I called it *tail-recursion.* For `g:A->B`, `h:B#SYMBOLS->B` **TRK(g,h)** is the function `f:A#WORDS->B` defined by `f(x,ε) = g(x)` and `f(x,w*a) = h(f(x,w),a)` for `x∈A`, `a∈SYMBOLS` and `w∈WORDS`. **h*** is the same for g being the identity function. A Nuprl-proof of these properties is performed in the theorems **trktype** - **trk4** in the library automata.

- **trktype**
  `THM >>∀A,B:TYPE. ∀g:A->B. ∀h:(B#SYMBOLS)->B. TRK(g,h) in (A#WORDS)->B`

---

* This is one of the advantages of defining objects by <u>not</u> using extract terms of theorems

* **trk1**
```
THM >>∀A,B:TYPE. ∀g:A->B. ∀h:(B#SYMBOLS)->B. ∀x:A. TRK(g,h)(x,ε) = g(x) in B
```

* **trk2**
```
THM >>∀A,B:TYPE. ∀g:A->B. ∀h:(B#SYMBOLS)->B. ∀x:A. ∀w:WORDS. ∀a:SYMBOLS
     . TRK(g,h)(x,w*a) = h(TRK(g,h)(x,w),a) in B
```

* **trk3**
```
THM >>∀A,B:TYPE. ∀g:A->B. ∀h:(B#SYMBOLS)->B. ∀x:A. ∀w:WORDS. TRK(g,h)(x,w) = h*(g(x),w) in B
```

* **trk4**
```
THM >>∀B:TYPE. ∀h:(B#SYMBOLS)->B. ∀u,v:WORDS. ∀x:B. h*(x,(u∘v)) = h*(h*(x,u),v) in B
```

All the formation and introduction rules for the type **WORDS** (i.e. rules dealing with uninteresting membership subgoals) were connected to one rule **wintro** which is easier to remember than all the individual rulenames. It is also used to expand the membership tactic to knowledge about **WORDS** which results in the very powerful tactic **Wmember** where **membertac** uses **wintro** as special type tactic before general membership strategies are applied. The same technique should be used for the other type-extensions too.

Elimination rules applied to words actually result in proofs by induction. In addition to the ordinary (head-) induction which comes from list-elimination two other types of induction proofs are sometimes very useful. This is induction on the tail of a word and induction over the length of a string.

```
H, w:WORDS, H'  >> T                                              word_elim hyp
   >> T[ε/w]
   hd:SYMBOLS, tl:WORDS, tl_hyp:T[tl/w] >> T[hd.tl/w]

H, w:WORDS, H'  >> T                                              word_elim_tail  hyp
   u:WORDS >> T[u/w] in U10
   >> T[ε/w]
   a:SYMBOLS, v:WORDS, T[v/w] >> T[v*a/w]

H, w:WORDS, H'  >> T                                              word_elim_lg  hyp
   u:WORDS >> T[u/w] in U10
   >> T[ε/w]
    i:int, 0<i, (∀v:WORDS. |v|=i-1 in int => T[v/w]), u:WORDS, |u|=i in int
   >> T[u/w]
```

The latter rules were written as tactics using the Nuprl-theorems **ind_principle** and **ind_principle1**.

* **ind_principle**
```
THM >>∀P:WORDS->U10. ( P(ε) & ∀b:SYMBOLS. ∀z:WORDS. P(z)=>P(z*b) )  =>  ∀w:WORDS.P(w)
```

* **ind_principle1**
```
THM >>∀P:WORDS->U10
     .(P(ε) &  ∀i:int.0<i => (∀z:WORDS.|z|=i-1 in int => P(z)) => (∀z:WORDS.|z|=i in int => P(z)))
     => ∀w:WORDS. P(w)
```

Since in Nuprl the level of a universe always has to be specified (no parameters possible), I had to choose a universe which seemed reasonably high. In my proofs so far I never had to go higher than **U2**. Therefore **U10** seemed to be high enough.

Using Nuprl-theorems in tactics is generally a very useful technique. It helps hiding the actual implementation of a new object such that only its properties have to be considered. Depending on the size of the supporting theorem it also shortens the proof actually performed by a tactic, i.e. the tactic will run faster. Many of the computation rules dealing with strings use this technique. The PRL extension to strings therefore comes with a huge collection of theorems and computation rules resulting from it which describe the basic laws of operations on strings. They cover trivial properties like concatenation with the empty word

```
* eps2
  THM >>∀v:WORDS. v∘ε = v in WORDS
```

which yields to the rule

```
  H >> v∘ε = w in words by wreduce 1                          eps_concat_right 1
     >> v = w in words
```

or the law of associativity of concatenation

```
* conassoz
  THM >> ∀u,v,w:WORDS. ((u∘v)∘w) = (u∘(v∘w)) in WORDS

  H >> (u∘v)∘z = w in words by wreduce 1                       con_assoz 1
     >> u∘(v∘v) = w in words
     >> u in words
     >> v in words
     >> w in words
```

the reverse of a concatenated word

```
* revcon
  THM >>∀u,v:WORDS. ←((u∘v)) = (←v∘←u) in WORDS

  H >> ←(u∘v) = w in words by wreduce 1                        rev_con 1
     >> ←v∘←u = w in words
     >> u in words
     >> v in words
```

a method to prove that a word is not empty if one just knows its length or vice versa

```
* lgprop1
  THM >>∀w:WORDS. w≠ε <=> 0<|w|
```

and many laws dealing with concatenation of substrings of a word, for example

```
* rangethm
  THM >>∀w:WORDS. ∀l,r:int. 0≤l & l≤r & r≤|w| =>  w = (w[1..l]∘w[l+1..r])∘w[r+1..lg] in WORDS
```

Most of these definitions are intuitively clear but the proof is not always simple. It often depends on other properties which previously had to be proven. Appendix A, chapter 1.3 gives a complete list of all these theorems. Not all of them are formulated as rules yet (e.g. **rangethm**). Like the introduction rules all the computation rules for **WORDS** were connected into a single rule **wreduce**. Thus the individual rulenames become unimportant for a user unless he intends to write efficient tactics.

The tactics written for **WORDS** deal with all-introduction (e.g. **word_all_intro**), induction on **WORDS** (i.e. all-introduction and then elimination of the introduced word), repeated introduction of words for existential formulas (**word_some_intro**) and two special cases as described in appendix B under `words_tactics.ml`.

4.Deterministic finite Automata

Given all the preparations the definitions for deterministic finite automata are straightforward now.

```
STATES                              == FINITE SETS
DFA                                 == (Q:STATES # ( ({Q}#SYMBOLS)->{Q} ) # {Q} # P({Q}))

<del:QxSYMBOLS-\>Q>*                == TRK((\x.x),<del>)
<word:WORDS> ε L(<AUT:DFA>)         == d*(q0,<word>) ε {F}- where (Q,d,q0,F)=<AUT>-
L(<AUTOM:DFA>)                      == {w:WORDS|w ε L(<AUTOM>)}
```

Again there is a lot of rules which deal with formation, introduction, elimination and computation. Their purpose is often easy to understand if one looks at the applications within the proof of the pumping_lemma. This proof is the first "big" Nuprl proof of a theorem which also has some interesting computational consequences. Its formalization looks very similar to the one found in computer science textbooks.

```
* pumping
  THM >>∀M:DFA. ∃n:int. ∀z:WORDS. zεL(M) & n≤|z|
       => ∃u,v,w:WORDS
      . z=(u∘v)∘w in WORDS & |u∘v|≤n & 0<|v| & ∀k:int.(u∘(v↑k))∘wεL(M)
```

I preferred the formulation **∀z:WORDS.z ε L(M) =>..** to **∀z:L(M). ..** because otherwise the property **d*(q0,z) ε{F} -where (Q,d,q0,F) = M-** would stay hidden in the proof too long. This shows one of the strong disadvantges of the "nonconstructive" set-type of PRL. The theorem was reproved several times until the proof reached a form which strongly resembles the original one. Planning the proof steps carefully and combining several steps into a larger and meaningful one was the most important technique I used. Sometimes the method how to reach a certain subgoal within one step was planned in front of the screen instead by looking into the manual respectively my tactics descriptions. It is often faster to proceed in this way. I took one wellformedness-goal out of the main proof and formulated it as a seperate lemma. The reason was that several versions of it occured as subgoals and that there was no way to avoid this. The tactic **THEOREM** now could take care of this problem which was rather PRL-specific.

```
* pumping_type
  THM >>∀M:DFA. ∀n:int. ∀z:WORDS. ∀u,v,w:WORDS
       . ( z=(u∘v)∘w in WORDS & |u∘v|≤n & 0<|v| & ∀k:int.(u∘(v↑k))∘wεL(M) ) in U1
```

The result is a proof which can be presented here in full detail.

In the first step the automaton `M` is introduced and split up into its parts. This is done by a combined tactic **dfa_intro_elim** which also takes care of wellformedness problems (e.g. `DFA in U2`). After that the critical number `n` is set to be the number of states in `M` and the PRL-specific subgoals (`n` is actually an integer, the rest of the expression is wellformed) are proven using the theorem **pumping_type**.

```
+------------------------------------------------------------------------------+
|EDIT THM pumping                                                              |
|------------------------------------------------------------------------------|
|* top                                                                         |
|>> ∀M:DFA. ∃n:int. ∀z:WORDS. z ε L(M) & n≤|z|                                 |
|     => ∃u,v,w:WORDS                                                          |
|   . z=((u∘v)∘w) in WORDS & |(u∘v)|≤n & 0<|v| & ∀k:int. ((u∘(v↑k))∘w) ε L(M)  |
|                                                                              |
|BY (* Let M = (Q,d,q0,F) be a deterministic finite automaton *)  dfa_intro_elim|
|                                                                              |
|1* 1. M:DFA                                                                   |
|   2. Q:STATES                                                                |
|   3. d:({Q}#SYMBOLS)->{Q}                                                    |
|   4. q0:{Q}                                                                  |
|   5. F:P({Q})                                                                |
|   6. (Q,d,q0,F)=M in DFA                                                     |
|   >>∃n:int. ∀z:WORDS. z ε L(Q,d,q0,F) & n≤|z|                                |
|      => ∃u,v,w:WORDS                                                         |
|    .z=((u∘v)∘w) in WORDS & |(u∘v)|≤n & 0<|v| & ∀k:int.((u∘(v↑k))∘w)εL(Q,d,q0,F)|
|                                                                              |
+------------------------------------------------------------------------------+
```

```
+------------------------------------------------------------------------------+
|EDIT THM pumping                                                              |
|------------------------------------------------------------------------------|
|* top 1                                                                       |
|1. M:DFA                                                                      |
|2. Q:STATES                                                                   |
|3. d:({Q}#SYMBOLS)->{Q}                                                       |
|4. q0:{Q}                                                                     |
|5. F:P({Q})                                                                   |
|6. (Q,d,q0,F)=M in DFA                                                        |
|>> ∃n:int. ∀z:WORDS. z ε L(Q,d,q0,F) & n≤|z|                                  |
|     => ∃u,v,w:WORDS                                                          |
|   .z=((u∘v)∘w) in WORDS & |(u∘v)|≤n & 0<|v| & ∀k:int.((u∘(v↑k))∘w)εL(Q,d,q0,F)|
|                                                                              |
|BY (* choose  n  to be the number of states in Q  *)                          |
|                                                                              |
|   Some_intro 'Q'                                                             |
|                                                                              |
|   membership                         (* shows that Q actually is an integer *)|
|                                                                              |
|   (Intro THENL                       (* show wellformedness of the goal     *)|
|    [ word_equality                                                           |
|    ; Intro THENL                                                             |
|      [ Intro THENL [-accepted_intro ; Wmember]                               |
|      ; Intro THENL                                                           |
|        [wequal; Intro THENL [wequal;Intro THENL [wequal;THEOREM pumping_type]]]|
|    ] ]                                                                       |
|   )                                                                          |
|                                                                              |
|1* >> ∀z:WORDS. z ε L(Q,d,q0,F) & (Q)≤|z|                                     |
|       => ∃u,v,w:WORDS                                                        |
|     .z=((u∘v)∘w) in WORDS & |(u∘v)|≤(Q) & 0<|v| & ∀k:int.((u∘(v↑k))∘w)εL(Q,d,q0,F)|
|                                                                              |
+------------------------------------------------------------------------------+
```

Now a string z is chosen that fulfills the required properties. These are moved from the conclusion into the hypothesis list. It is easy to predict the hypothesis number 8 where the conjunction `zεL(Q,d,q0,F) & (Q)≤|z|` will be found after the introduction steps. The tactic **membership** together with a dfa-specific tactic deals with the wellformedness subgoals.

```
+------------------------------------------------------------------------------+
|EDIT THM pumping                                                              |
|------------------------------------------------------------------------------|
|* top 1 1                                                                     |
|1. M:DFA                                                                      |
|2. Q:STATES                                                                   |
|3. d:({Q}#SYMBOLS)->{Q}                                                       |
|4. q0:{Q}                                                                     |
|5. F:P({Q})                                                                   |
|6. (Q,d,q0,F)=M in DFA                                                        |
|>> ∀z:WORDS. z ε L(Q,d,q0,F) & (Q)≤|z|                                        |
|     => ∃u,v,w:WORDS                                                          |
|    .z=((u∘v)∘w) in WORDS & |(u∘v)|≤(Q) & 0<|v| & ∀k:int.((u∘(v↑k))∘w) εL(Q,d,q0,F) |
|                                                                              |
|BY (* Let z be a string such that z ε L(M) and n ≤ |z| *)                     |
|                                                                              |
|    word_all_intro     THEN Intro                                             |
|    THENL [and_elim 8; Intro THENL [-accepted_intro; membership] ]            |
|                                                                              |
|1* 7. z:WORDS                                                                 |
|   8. z ε L(Q,d,q0,F)                                                         |
|   9. (Q)≤|z|                                                                 |
|   >> ∃u,v,w:WORDS                                                            |
|    .z=((u∘v)∘w) in WORDS & |(u∘v)|≤(Q) & 0<|v| & ∀k:int.((u∘(v↑k))∘w) εL(Q,d,q0,F) |
|                                                                              |
+------------------------------------------------------------------------------+
```

Since z is supposed to be longer than n there must be a loop in the transition-diagram, i.e. some `i<j≤n` such that `d*(q0,z[1..i]) = d*(q0,(z[1..i]∘z[i+1..j]))`. This fact is sequenced in and unraveled for further use. Two subgoals are to prove now.

```
+------------------------------------------------------------------------------+
|EDIT THM pumping                                                              |
|------------------------------------------------------------------------------|
|* top 1 1 1                                                                   |
|1. M:DFA                                                                      |
|2. Q:STATES                                                                   |
|3. d:({Q}#SYMBOLS)->{Q}                                                       |
|4. q0:{Q}                                                                     |
|5. F:P({Q})                                                                   |
|6. (Q,d,q0,F)=M in DFA                                                        |
|7. z:WORDS                                                                    |
|8. z ε L(Q,d,q0,F)                                                            |
|9. (Q)≤|z|                                                                    |
|>> ∃u,v,w:WORDS                                                               |
|    .z=((u∘v)∘w) in WORDS & |(u∘v)|≤(Q) & 0<|v| & ∀k:int.((u∘(v↑k))∘w) εL(Q,d,q0,F) |
|                                                                              |
|BY (* Since z is longer than n there must be a loop in the transition diagram, |
|      i.e. some i < j ≤ n such that d*(q0,z[1..i]) = d*( q0,(z[1..i]∘z[i+1..j]) ) *) |
|                                                                              |
|    Seq ['∃i,j:int. 0≤i & i<j & j≤Q & d*(q0,z[1..i])=d*(q0,(z[1..i]∘z[i+1..j])) in {Q}'] |
|    THENL [IDTAC; some_elim 10 THEN (and_elim 12)]                            |
|                                                                              |
|1* >> ∃i,j:int. 0≤i & i<j & j≤Q & d*(q0,z[1..i])=d*(q0,(z[1..i]∘z[i+1..j])) in {Q} |
|                                                                              |
|2* 10. i:int                                                                  |
|   11. j:int                                                                  |
|   12. d*(q0,z[1..i]) = d*(q0,(z[1..i]∘z[i+1..j])) in {Q}                     |
|   13. 0≤i                                                                    |
|   14. i<j                                                                    |
|   15. j≤Q                                                                    |
|   >> ∃u,v,w:WORDS                                                            |
|    .z=((u∘v)∘w) in WORDS & |(u∘v)|≤(Q) & 0<|v| & ∀k:int.((u∘(v↑k))∘w) εL(Q,d,q0,F) |
|                                                                              |
+------------------------------------------------------------------------------+
```

The first subgoal follows from an application of the pigeonhole as formulated in **pigeon_variant**. This part of the proof is rather uninteresting for the proof itself because it mainly deals with PRL-specific problems. I therefore put its proof together into a single "rule". By using the tactic **THEOREM1** and some elimination and computation its first step (in parentheses) generates the new hypotheses

```
k:int, l:int, 0≤k, k<l, l≤n, d*(q0,z[1..k]) = d*(q0,z[1..l]) in {Q}
```

and the rest of this rule is used to apply this in the conclusion, transform `z[1..l]` into `z[1..k]∘z[k+1..l]` and - again - deal with wellformedness.

```
+----------------------------------------------------------------------------+
|EDIT THM pumping                                                            |
|----------------------------------------------------------------------------|
|* top 1 1 1 1                                                               |
|1. M:DFA                                                                    |
|2. Q:STATES                                                                 |
|3. d:({Q}#SYMBOLS)->{Q}                                                     |
|4. q0:{Q}                                                                   |
|5. F:P({Q})                                                                 |
|6. (Q,d,q0,F)=M in DFA                                                      |
|7. z:WORDS                                                                  |
|8. z ε L(Q,d,q0,F)                                                          |
|9. (Q)≤|z|                                                                  |
|>> ∃i,j:int. 0≤i & i<j & j≤Q & d*(q0,z[1..i])=d*(q0,(z[1..i]∘z[i+1..j])) in{Q}|
|                                                                            |
|BY (*     apply the pigeonhole principle     *)                             |
|                                                                            |
|   (-THEOREM1 `pigeon_variant` ['Q';'(\n.d*(q0,z[1..n]))']                  |
|    THENL                                                                   |
|    [ Intro THENL [@delstar_apply; membership]                              |
|    ; some_elim 10                THEN and_elim 12                          |
|      THEN Hyp_compute_term 12 1  THEN Hyp_compute_snd_term 12 1            |
|    ]                                                                       |
|   )                                                                        |
|   THEN (      repeat_some_intro 1 ['k';'l']                                |
|         THENL [Intro;Intro; IDTAC; Equal;Equal; -repeat_and_intro] )       |
|   THENL                                                                    |
|   [ Intro                                                                  |
|     THENL [membership; Intro THENL [-state_set_intro;delstar_apply;delstar_apply]]|
|   ; -SubstFor '(z[1..k]∘z[k+1..l]) = z[1..l] in WORDS'                     |
|     THENL                                                                  |
|     [THEOREM range9; Intro THENL [-state_set_intro;delstar_apply;delstar_apply]]|
|   ]                                                                        |
|                                                                            |
+----------------------------------------------------------------------------+
```

In the second subgoal the proof now is ready for the introduction of the desired three substrings. A special variant of the tactic **repeat_some_intro**, which spared me of a lot of unnecessary wellformedness-subgoals, was used for that. The theorem **pumping_type** therefore had to be applied only once. `u = z[1..i]`, `v = z[i+1..j]` and `w = z[j+1..lg]` were chosen. The theorem requires them to fulfill four conditions which result in the four subgoals of this step.

```
+---------------------------------------------------------------------------------+
|EDIT THM pumping                                                                 |
|---------------------------------------------------------------------------------|
|* top 1 1 1 2                                                                    |
|1. M:DFA                                                                         |
|2. Q:STATES                                                                      |
|3. d:({Q}#SYMBOLS)->{Q}                                                          |
|4. q0:{Q}                                                                        |
|5. F:P({Q})                                                                      |
|6. (Q,d,q0,F)=M in DFA                                                           |
|7. z:WORDS                                                                       |
|8. z ε L(Q,d,q0,F)                                                               |
|9. (Q)≤|z|                                                                       |
|10. i:int                                                                        |
|11. j:int                                                                        |
|12. d*(q0,z[1..i]) = d*(q0,(z[1..i]∘z[i+1..j])) in {Q}                           |
|13. 0≤i                                                                          |
|14. i<j                                                                          |
|15. j≤Q                                                                          |
|>> ∃u,v,w:WORDS                                                                  |
|   .z=((u∘v)∘w) in WORDS & |(u∘v)|≤(Q) & 0<|v| & ∀k:int.((u∘(v↑k))∘w) εL(Q,d,q0,F) |
|                                                                                 |
|BY (* choose  u=z[1..i],  v=z[i+1..j],  w=z[j+1..lg]     *)                      |
|                                                                                 |
|   word_some_intro 1 ['z[1..i]';'z[i+1..j]';'z[j+1..lg]']                        |
|   THENL             [ THEOREM pumping_type; repeat_and_intro ]                  |
|                                                                                 |
|1* >> z = ((z[1..i]∘z[i+1..j])∘z[j+1..lg]) in WORDS                              |
|                                                                                 |
|2* >> |(z[1..i]∘z[i+1..j])|≤(Q)                                                  |
|                                                                                 |
|3* >> 0 < |z[i+1..j]|                                                            |
|                                                                                 |
|4* >> ∀k:int. ((z[1..i]∘(z[i+1..j]↑k))∘z[j+1..lg]) ε L(Q,d,q0,F)                 |
|                                                                                 |
+---------------------------------------------------------------------------------+
```

The proofs of first three subgoals are obvious and need no further comment. They mainly follow from theorems alrady proven.

```
+---------------------------------------------------------------------------------+
|EDIT THM pumping                                                                 |
|---------------------------------------------------------------------------------|
|* top 1 1 1 2 1                                                                  |
|1. M:DFA                                                                         |
|2. Q:STATES                                                                      |
|3. d:({Q}#SYMBOLS)->{Q}                                                          |
|4. q0:{Q}                                                                        |
|5. F:P({Q})                                                                      |
|6. (Q,d,q0,F)=M in DFA                                                           |
|7. z:WORDS                                                                       |
|8. z ε L(Q,d,q0,F)                                                               |
|9. (Q)≤|z|                                                                       |
|10. i:int                                                                        |
|11. j:int                                                                        |
|12. d*(q0,z[1..i]) = d*(q0,(z[1..i]∘z[i+1..j])) in {Q}                           |
|13. 0≤i                                                                          |
|14. i<j                                                                          |
|15. j≤Q                                                                          |
|>> z = ((z[1..i]∘z[i+1..j])∘z[j+1..lg]) in WORDS                                 |
|                                                                                 |
|BY (* Clear by the definition of u,v and w *)           THEOREM rangethm         |
|                                                                                 |
+---------------------------------------------------------------------------------+
```

```
+-----------------------------------------------------------------------------+
|EDIT THM pumping                                                             |
|-----------------------------------------------------------------------------|
|* top 1 1 1 2 2                                                              |
|1. M:DFA                                                                     |
|2. Q:STATES                                                                  |
|3. d:({Q}#SYMBOLS)->{Q}                                                      |
|4. q0:{Q}                                                                    |
|5. F:P({Q})                                                                  |
|6. (Q,d,q0,F)=M in DFA                                                       |
|7. z:WORDS                                                                   |
|8. z ε L(Q,d,q0,F)                                                           |
|9. (Q)≤|z|                                                                   |
|10. i:int                                                                    |
|11. j:int                                                                    |
|12. d*(q0,z[1..i]) = d*(q0,(z[1..i]∘z[i+1..j])) in {Q}                       |
|13. 0≤i                                                                      |
|14. i<j                                                                      |
|15. j≤Q                                                                      |
|>> |(z[1..i]∘z[i+1..j])|≤(Q)                                                 |
|                                                                             |
|BY (* |u∘v| = j ≤ Q   *)                                                     |
|                                                                             |
|          @SubstFor '|(z[1..i]∘z[i+1..j])| = j in int'                       |
|   THEN    SubstFor '(z[1..i]∘z[i+1..j]) = z[1..j] in WORDS'                 |
|   THENL  [ THEOREM range9; THEOREM lgsuf THEN &Arith; Wmember]              |
|                                                                             |
+-----------------------------------------------------------------------------+
```

```
+-----------------------------------------------------------------------------+
|EDIT THM pumping                                                             |
|-----------------------------------------------------------------------------|
|* top 1 1 1 2 3                                                              |
|1. M:DFA                                                                     |
|2. Q:STATES                                                                  |
|3. d:({Q}#SYMBOLS)->{Q}                                                      |
|4. q0:{Q}                                                                    |
|5. F:P({Q})                                                                  |
|6. (Q,d,q0,F)=M in DFA                                                       |
|7. z:WORDS                                                                   |
|8. z ε L(Q,d,q0,F)                                                           |
|9. (Q)≤|z|                                                                   |
|10. i:int                                                                    |
|11. j:int                                                                    |
|12. d*(q0,z[1..i]) = d*(q0,(z[1..i]∘z[i+1..j])) in {Q}                       |
|13. 0≤i                                                                      |
|14. i<j                                                                      |
|15. j≤Q                                                                      |
|>> 0 < |z[i+1..j]                                                            |
|                                                                             |
|BY (* |v| = j-i > 0   *)                                                     |
|                                                                             |
|   SubstFor '|z[i+1..j]|=j-i in int'                                         |
|   THENL    [THEOREM lgrange; THEOREM lessarith; &Cumulativity]              |
|                                                                             |
+-----------------------------------------------------------------------------+
```

In the fourth subgoal the definition of `zεL(M)` is transformed into its original form `d*(q0,z)ε{F}` and the same is done to the conclusion after the introduction of `k`.

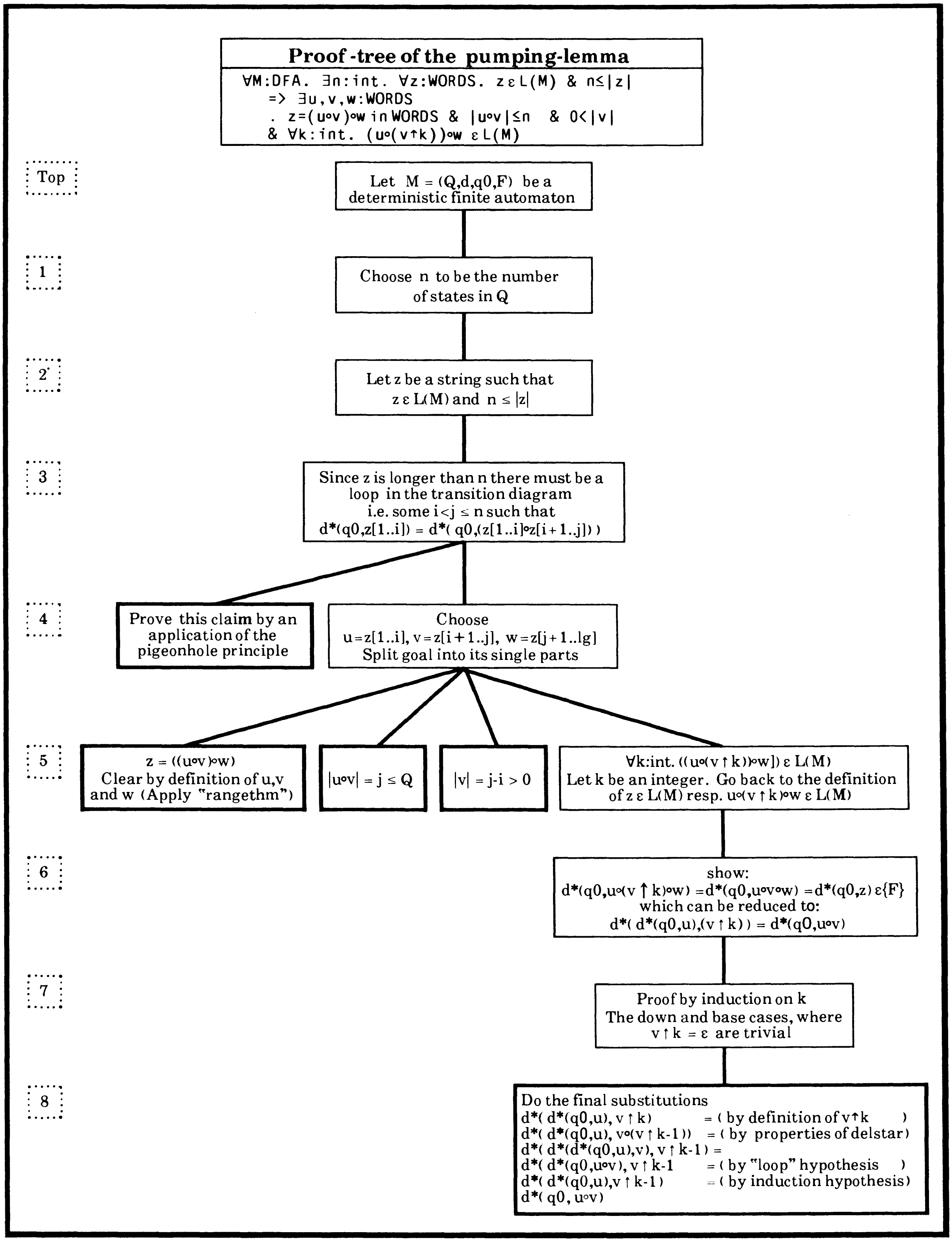
```
+----------------------------------------------------------------------------+
|EDIT THM pumping                                                            |
|----------------------------------------------------------------------------|
|* top 1 1 1 2 4                                                             |
|1. M:DFA                                                                    |
|2. Q:STATES                                                                 |
|3. d:({Q}#SYMBOLS)->{Q}                                                     |
|4. q0:{Q}                                                                   |
|5. F:P({Q})                                                                 |
|6. (Q,d,q0,F)=M in DFA                                                      |
|7. z:WORDS                                                                  |
|8. z ε L(Q,d,q0,F)                                                          |
|9. (Q)≤|z|                                                                  |
|10. i:int                                                                   |
|11. j:int                                                                   |
|12. d*(q0,z[1..i]) = d*(q0,(z[1..i]∘z[i+1..j])) in {Q}                      |
|13. 0≤i                                                                     |
|14. i<j                                                                     |
|15. j≤Q                                                                     |
|>> ∀k:int. ((z[1..i]∘(z[i+1..j]↑k))∘z[j+1..lg]) ε L(Q,d,q0,F)               |
|                                                                            |
|BY (* Let k be an integer. Go back to the definition of zεL(M) resp. u∘(v↑k)∘wεL(M) *)|
|                                                                            |
|   int_all_intro  THEN Compute 3  THEN Hyp_compute 8 3                      |
|                                                                            |
|1* 8. d*(q0,z) ε {F}                                                        |
|   16. k:int                                                                |
|   >> d*(q0,(z[1..i]∘(z[i+1..j]↑k))∘z[j+1..lg])) ε {F}                      |
|                                                                            |
+----------------------------------------------------------------------------+
```

The proof now follows immediately if `d*(q0,u∘(v↑k)∘w) = d*(q0,u∘v∘w) = d*(q0,z)` can be shown. This again can be reduced to the problem `d*( d*(q0,u),(v↑k) ) = d*(q0,u∘v)` which is a consequence of hypothesis 12. The rest of the proof shows this by induction on `k` and by a series of substitutions.

```
+----------------------------------------------------------------------------+
|EDIT THM pumping                                                            |
|----------------------------------------------------------------------------|
|* top 1 1 1 2 4 1                                                           |
|1. M:DFA                                                                    |
|2. Q:STATES                                                                 |
|3. d:({Q}#SYMBOLS)->{Q}                                                     |
|4. q0:{Q}                                                                   |
|5. F:P({Q})                                                                 |
|6. (Q,d,q0,F)=M in DFA                                                      |
|7. z:WORDS                                                                  |
|8. d*(q0,z) ε {F}                                                           |
|9. (Q)≤|z|                                                                  |
|10. i:int                                                                   |
|11. j:int                                                                   |
|12. d*(q0,z[1..i]) = d*(q0,(z[1..i]∘z[i+1..j])) in {Q}                      |
|13. 0≤i                                                                     |
|14. i<j                                                                     |
|15. j≤Q                                                                     |
|16. k:int                                                                   |
|>> d*(q0,((z[1..i]∘(z[i+1..j]↑k))∘z[j+1..lg])) ε {F}                        |
|                                                                            |
|BY (* show:                    d*(q0,u∘(v↑k)∘w)     =d*(q0,u∘v∘w)=d*(q0,z)ε{F} |
|    which can be reduced to:  d*( d*(q0,u),(v↑k) ) = d*(q0,u∘v)          *) |
|                                                                            |
|   -SubstFor 'd*(q0,((z[1..i]∘(z[i+1..j]↑k))∘z[j+1..lg])) = d*(q0,z) in {Q}'|
|   THENL                                                                    |
|   [ SubstFor 'd*(q0,((z[1..i]∘(z[i+1..j]↑k))∘z[j+1..lg]))                  |
|            = d*(q0,((z[1..i]∘z[i+1..j])∘z[j+1..lg]))    in {Q}'            |
|     THENL                                                                  |
|     [ star_concat_compute 1       THEN star_concat_compute 2   THEN delstar_apply |
|       THEN star_concat_compute 2  THEN Thinning [1;5;6;8;9;13;15]          |
|     ; delstar_apply THEN theorem `rangethm` ['z';'i';'j';imp]              |
|     ; Cumulativity  THEN -Intro THENL [-state_set_intro; delstar_apply]    |
|     ]                                                                      |
|   ; Elim 5 THENL [membership; -Sinset]                                     |
|   ]                                                                        |
|                                                                            |
|1* >>d*(d*(q0,z[1..i]),(z[i+1..j]↑k)) = d*(q0,(z[1..i]∘z[i+1..j])) in {Q}   |
+----------------------------------------------------------------------------+
```

```
EDIT THM pumping
--------------------------------------------------------------------------------
* top 1 1 1 2 4 1 1
1. Q:STATES
2. d:({Q}#SYMBOLS)->{Q}
3. q0:{Q}
4. z:WORDS
5. i:int
6. j:int
7. d*(q0,z[1..i]) = d*(q0,(z[1..i]∘z[i+1..j])) in {Q}
8. i<j
9. k:int
>> d*(d*(q0,z[1..i]),(z[i+1..j]↑k))=d*(q0,(z[1..i]∘z[i+1..j])) in {Q}

BY (*  Induction on k. The down and base cases, where v↑k = ε are
       trivial because they follow directly from hypotheses 7      *)

   Int_elim 9
   THENL
   [ SubstFor 'z[i+1..j]↑m = ε in WORDS'
     THENL
     [ -iter_down 1 THEN weps
     ; -star_reduction_eps 1
     ; Cumulativity THEN Intro THENL [-state_set_intro;delstar_apply;delstar_apply]
     ]
   ; Compute_apply2_arg2 1 THEN -star_reduction_eps 1
   ; IDTAC
   ]

1* 10. m:int
   11. 0<m
   12. hyp: d*( d*(q0,z[1..i]),(z[i+1..j]↑(m-1)) ) = d*(q0,(z[1..i]∘z[i+1..j])) in {Q}
   >>d*(d*(q0,z[1..i]),(z[i+1..j]↑m) =d*(q0,(z[1..i]∘z[i+1..j])) in {Q}
```

```
EDIT THM pumping
--------------------------------------------------------------------------------
* top 1 1 1 2 4 1 1 1
1. Q:STATES
2. d:({Q}#SYMBOLS)->{Q}
3. q0:{Q}
4. z:WORDS
5. i:int
6. j:int
7. d*(q0,z[1..i]) = d*(q0,(z[1..i]∘z[i+1..j])) in {Q}
8. i<j
9. k:int
10. m:int
11. 0<m
12. hyp: d*(d*(q0,z[1..i]),(z[i+1..j]↑(m-1))) = d*(q0,(z[1..i]∘z[i+1..j])) in {Q}
>> d*(d*(q0,z[1..i]),(z[i+1..j]↑m) = d*(q0,(z[1..i]∘z[i+1..j])) in {Q}

BY (* Do the final substitutions
      d*( d*(q0,u), v↑k)           =       ( by definition of v↑k  )
      d*( d*(q0,u), v∘(v↑k-1))     =       ( properties of delstar )
      d*( d*( d*(q0,u),v), v↑k-1) =
      d*( d*(q0,u∘v), v↑k-1)       =       ( hypothesis 7          )
      d*( d*(q0,u),v↑k-1)          =       ( induction hypothesis  )
      d*( q0, u∘v)                                                     *)

   SubstFor  'z[i+1..j]↑m = (z[i+1..j]∘z[i+1..j]↑(m-1)) in WORDS'
   THENL
   [ -iter_up 1 THEN Wmember
   ; star_concat_compute 1   THENL [membership; IDTAC]
   ; Cumulativity THEN Intro THENL [-state_set_intro;delstar_apply;delstar_apply]
   ]
   THEN  SubstFor 'd*( d*(q0,z[1..i]),z[i+1..j]) = d*(q0,z[1..i]) in {Q}'
   THENL
   [-THEOREM1 `delprop1` ['Q';'d';'z[1..i]';'z[i+1..j]';'q0'] THEN Wintro
   ; Equal
   ; Cumulativity THEN Intro THENL [-state_set_intro;@delstar_apply;delstar_apply]
   ]
```


Proof -tree of the pumping-lemma
∀M:DFA. ∃n:int. ∀z:WORDS. z ε L(M) & n≤|z|
=> ∃u,v,w:WORDS
. z=(u∘v)∘w in WORDS & |u∘v|≤n & 0<|v|
& ∀k:int. (u∘(v↑k))∘w ε L(M)
Top
Let M = (Q,d,q0,F) be a deterministic finite automaton
1
Choose n to be the number of states in Q
2
Let z be a string such that z ε L(M) and n ≤ |z|
3
Since z is longer than n there must be a loop in the transition diagram
i.e. some i<j ≤ n such that
d*(q0,z[1..i]) = d*(q0,(z[1..i]∘z[i+1..j]))
4
Prove this claim by an application of the pigeonhole principle
Choose
u=z[1..i], v=z[i+1..j], w=z[j+1..lg]
Split goal into its single parts
5
z = ((u∘v)∘w)
Clear by definition of u,v and w (Apply "rangethm")
|u∘v| = j ≤ Q
|v| = j-i > 0
∀k:int. ((u∘(v↑k))∘w]) ε L(M)
Let k be an integer. Go back to the definition of z ε L(M) resp. u∘(v↑k)∘w ε L(M)
6
show:
d*(q0,u∘(v↑k)∘w) =d*(q0,u∘v∘w) =d*(q0,z) ε{F}
which can be reduced to:
d*(d*(q0,u),(v↑k)) = d*(q0,u∘v)
7
Proof by induction on k
The down and base cases, where
v↑k = ε are trivial
8
Do the final substitutions
d*(d*(q0,u), v↑k) = (by definition of v↑k)
d*(d*(q0,u), v∘(v↑k-1)) = (by properties of delstar)
d*(d*(d*(q0,u),v), v↑k-1) =
d*(d*(q0,u∘v), v↑k-1 = (by "loop" hypothesis)
d*(d*(q0,u),v↑k-1) = (by induction hypothesis)
d*(q0, u∘v)

The implementation of the pumping-lemma does not only give a mechanical proof of it. It also has a computational content. Given a deterministic finite automaton `M` and a string `z ε L(M)` which is long enough, the extract term of the theorem **pumping** actually computes the critical number `n` and the three substrings `u,v,w` of `z` which fulfill the conditions of the pumping-lemma.

The eval-object **pumping_demo** in the automata-theory library contains two examples of deterministic finite automata and the interesting part of the extract-term of the theorem pumping. It also contains an object `test` which causes the PRL-system to open all the theorems necessary for a demonstration while loading the library (which takes about an hour now). This is necessary because the current PRL implementation does not check (and evaluate) theorems unless they are used which saves time while loading a library.

```
+------------------------------------------------------------------------------+
|EDIT EVAL pumping_demo                                                        |
|------------------------------------------------------------------------------|
|(* Automaton for (11+22)*   *)                                                |
|let M = (4, λq,a.int_eq(a;1;int_eq(q;1;2;int_eq(q;2;1;4)));                   |
|                int_eq(a;2;int_eq(q;1;3;int_eq(q;3;1;4)); 4 ))               |
|         ,1,{4:{1,..,4}} )                                                    |
|;;                                                                            |
|                                                                              |
|(* Automaton for 2(112)* *)                                                   |
|let M1 = (5,λq,a.int_eq(a;1;int_eq(q;1;5;(1+q) mod 6);                        |
|                int_eq(a;2;int_eq(q;1;2;int_eq(q;4;2;5)); 5))                |
|          ,1,{2:{1,..,5}} )                                                   |
|;;                                                                            |
|                                                                              |
|(* Evaluation term of the pumping_lemma *)                                    |
|let pump   = λM,z.( n ,(u, v, w) - where (u,v,w,proof)=(pump(z)(axiom,axiom))- ) |
|               - where (n,pump)=term_of(pumping)(M)-                          |
|;;                                                                            |
|let test = pump( M , 1.1.2.2.1.1.ε );;                                        |
|                                                                              |
+------------------------------------------------------------------------------+
```

The following snapshot shows some examples of the evaluation of the pumping lemma on these two automata and several words. The computation of the results in each case took less than 2 seconds.

```
+------------------------------------------------------------------------------+
| NuPRL Command/Status                                                         |
|------------------------------------------------------------------------------|
|E>pump(< M , 1.1.2.2.1.1.ε >);;                                               |
|< 4 , < ε ,< 1.1.2.2.ε , 1.1.ε > > >                                          |
|E>pump(< M , 2.2.1.1.2.2.2.2.2.2.1.1.ε >);;                                   |
|< 4 , < ε,< 2.2.1.1.ε , 2.2.2.2.2.2.1.1.ε > > >                               |
|E>pump(< M1 , 2.1.1.2.1.1.2.ε >);;                                            |
|< 5 , < 2.1.ε , < 1.2.1.ε , 1.2.ε > > >                                       |
|E>pump(< M1 , 2.1.1.2.1.1.2.1.1.2.1.1.2.ε >);;                                |
|< 5 , < 2.1.ε , < 1.2.1.ε , 1.2.1.1.2.1.1.2.ε > > >                           |
|                                                                              |
+------------------------------------------------------------------------------+
```

5.Future plans

5.1 Expanding the automata theory library

In this chapter I will present some concepts how to implement more definitions and theorems of basic automata theory.

Nondeterministic finite automata *(nfa)* can be implemented similarly to deterministic finite automata. The definitions from conventional automata theory would have to be formalized as follows

```
NFA                              == (Q:STATES # ( ({Q}#SYMBOLS)->P({Q}) ) # {Q} # P({Q}))

<d:QxSYMBOLS-\>P(Q)>*            == TRK( \q.{q:{Q}}, λf,a.{p:{Q}|| ∃r:{f}. pε{<d>(r,a)} } )
<w:WORDS> ε L(<A:NFA>)           == ∃f:{d*(q0,<w>)}. f ε {F}- where (Q,d,q0,F)=<A>-
L(<A:NFA>)                       == {w:WORDS|w ε L(<A>)}
```

All the rules dealing with nondeterministic finite automata are essentially the same as the ones for dfa. The just have to be copied and slightly modified. However the Nuprl-proof of the equivalence between nfa and dfa

```
>> ∀M:NFA. ∃A:DFA. L(M) = L(A) in TYPE
```

requires some additional definitions which handle the simulation of `P({Q})` within the concept of finite sets and an extension `d+` of the transition function to `P({Q})`. The set `{1,..2↑Q}` could be used for that where `2↑Q` stands for the exponentiation of `Q`.

```
2↑<i:int>                                   == ind(<i>;x,y.1;1;x,two_exp_i.2*two_exp_i)
no(<Q:STATES>,<A:P({Q}) into {1,..2↑Q}>)    == sum(2↑i;iεA) *
set(<Q:STATES>,<n:{1,..2↑Q} into P({Q})>)   == {q:{<Q>}||(2↑q)-1 < n mod (2*(2↑q))}

d+(<Q:STATES>,<A:P({Q})>,<w:WORDS>)         == {q:{<Q>}|| ∃r:{<A>}. qε{d*(r,<w>)} }
```

Theorems and tactics have to be written which deal with the most important properties of these definitions, e.g. **`no(set(Q,n)) = n in {Q}`**, **`set(Q,no(A)) = A in P({Q})`**, **`Q:STATES >> 2↑Q in STATES`**, **`q:{Q} >> d+(Q,{q:{Q}},w} = d*(q,w) in {Q}`** etc. After that the proof of the equivalence theorem can easily be written in analogy to the original proof:

```
1. Let  M = (Q,d,q0,F) be a nfa

2. Choose A = (Q1,d1,q1,F1) =
   (2↑Q, λq,a.no(d+(Q,{set(Q,q)}.⊗a)), no({q0:Q}), {f:{2↑Q}|| ∃g:{set(Q,f)}.gε{F} } )
   Show that A is in DFA and that the rest of the expression is wellformed

3. Show ∀w:WORDS.∀p:Q1. d1*(p,w) = no( d+(set(Q,p),w) )  by tail-induction on w

4. Reduce L(M) = L(A) in TYPE using computation rules to
   ∃f:{d*(q0,w)}. f ε {F}  <=>  d1*(q1,w) in {f:{2↑Q} | ∃g:{set(Q,f)}. gε{F} }

5. Proof this by the following substitution sequence
           d1*(q1,w) in {f:{2↑Q} | ∃g:{set(Q,f)}. gε{F} }
       <=> no( d+(Q,set(Q,no({q0:Q})),w) ) in {f:{2↑Q} | ∃g:{set(Q,f)}. gε{F} }
       <=> no( d*(q0,w) in {f:{2↑Q} | ∃g:{set(Q,f)}. gε{F} }
       <=> ∃g:{set(Q,no( d*(q0,w))}. gε{F}
       <=> ∃g:{d*(q0,w)}. gε{F}
```

* It seems that for this conversion it is necessary that `A` is a `DECIDABLE SET`

The most interesting part of this theorem is that its extract_term will actually convert nondeterministic finite automata into the equivalent deterministic ones.

Nondeterministic finite Automata with ε-moves also have some similarity to dfa but they require some more formal work. While in this context CS-textbooks usually simply treat the empty string ε like a symbol, in an exact formalization a distinction has to be made. In Nuprl-proofs therefore often a case analysis (`w=ε` or `w≠ε`) which does not occur in the original proof is necessary. Definitions also become more complicated.

```
EPS                                == {w:WORDS| w=ε in WORDS}
ε-NFA                              == (Q:STATES # (({Q}#(SYMBOLS|EPS)->P({Q})) # {Q} # P({Q}))

ε-cl(<M:ε-NFA>,<n:N>,<q:{Q}>)      == ind(<n>; m,cl.{q:{Q}}; {q:{Q}};
                                      m,cl.{p:{Q}||∃r:{cl}.pε{d(r,inr(ε))}} -where (Q,d,q0,F)=M-)
ε-closure(<M:ε-NFA>,<q:{Q}>)       == {p:{Q}-where (Q,d,q0,F)=M-|| ∃n:N. pε ε-cl(M,n,p)}
ε-closure(<M:ε-NFA>,<A:P({Q})>)    == {p:{Q}-where (Q,d,q0,F)=M-|| ∃q:{A}. pε ε-closure(M,q)}

<d:QxSYMBOLS-\>P(Q)>* [<M:ε-NFA>]  == TRK(\q.ε-closure(<M>,q), λf,a.{p:{Q}
                                        -where (Q,d,q0,F)=<M>- || ∃r:{f}.pε{<d>(r,inl(a)} } )
<w:WORDS> ε L(<A:NFA>)             == ∃f:{d*[A](q0,<w>)}. f ε {F}- where (Q,d,q0,F)=<A>-
L(<A:NFA>)                         == {w:WORDS|w ε L(<A>)}
```

A Nuprl-proof of the equivalence between ε-nfa and nfa now can be written in analogy to the original one (See [1] for a proof structure). However it requires a lot of detailed work.

5.2 Modules

PRL-extension modules should provide a tool to actually use the extensions implemented so far. They are necessary to keep individual libraries small instead of having one big library. A module should consist of a piece of library which includes the necessary definitions and theorems and ML-functions (which usually are kept in a separate file) and - of course - a description as well. This means a extension might be used by just loading this library piece into the own library. The effect would be the same as if a new type (e.g. **WORDS**) would have been added to PRL. This however is under user control and subject to improvements.

Some of the extensions described above (`general tactics`, `logic`, `WORDS`) might be used the way they are (maybe some additions should be made) while others (`sets`, `finite sets`) should be reconsidered and worked out in greater detail before being made public.

Appendix A: The complete library

```
+--------------------------------------------------------------------+
| Library   Automata                                                 |
|--------------------------------------------------------------------|
|                                                                    |
| * AUTOMATA_THEORY                                                  |
|   DEF PRL-Version by Christoph Kreitz   February 25, 1986          |
|                                                                    |
|                                                                    |
| * c1                                                               |
|   DEF                                                              |
|            CHAPTER 1:     PRELIMINARIES                            |
|            ----------------------------                            |
|                                                                    |
| * c10                                                              |
|   DEF                                                              |
|            1.0:    general definitions for tactics                 |
|            ----                                                    |
| * c                                                                |
|   DEF (*<comment>*)                                                |
|                                                                    |
| * tac                                                              |
|   DEF -<tactic>                                                    |
|                                                                    |
| * tacm                                                             |
|   DEF @<tactic>                                                    |
|                                                                    |
| * tacmember                                                        |
|   DEF &<tactic>                                                    |
|                                                                    |
| * theorem                                                          |
|   DEF THEOREM <theorem_name>                                       |
|                                                                    |
|                                                                    |
| * c11                                                              |
|   DEF                                                              |
|            1.1:    NOTATION: logic definitions, tupling, functions |
|            ----                                                    |
|                                                                    |
| * and                                                              |
|   DEF <prop> & <prop>                                              |
|                                                                    |
| * all                                                              |
|   DEF ∀<var>:<type>. <prop>                                        |
|                                                                    |
| * all2                                                             |
|   DEF ∀<var>,<var>:<type>. <prop>                                  |
|                                                                    |
| * all3                                                             |
|   DEF ∀<var>,<var>,<var>:<type>. <prop>                            |
|                                                                    |
| * some                                                             |
|   DEF ∃<var>:<type>. <prop>                                        |
|                                                                    |
| * some2                                                            |
|   DEF ∃<var>,<var>:<type>. <prop>                                  |
|                                                                    |
| * some3                                                            |
|   DEF ∃<var>,<var>,<var>:<type>. <prop>                            |
|                                                                    |
| * imp                                                              |
|   DEF <prop> => <prop>                                             |
|                                                                    |
| * equiv                                                            |
|   DEF <prop> <=> <prop>                                            |
|                                                                    |
| * false                                                            |
|   DEF false                                                        |
|                                                                    |
| * not                                                              |
|   DEF ¬<prop>                                                      |
|                                                                    |
| * type                                                             |
|   DEF TYPE                                                         |
|                                                                    |
| * tup2                                                             |
|   DEF (<term>,<term>)                                              |
|                                                                    |
| * tup3                                                             |
|   DEF (<term>,<term>,<term>)                                       |
|                                                                    |
| * tup4                                                             |
|   DEF (<term>,<term>,<term>,<term>)                                |
|                                                                    |
```

```
* detup2
  DEF <term> -where (<id>,<id>)=<term>-

* detup3
  DEF <term>- where (<id>,<id>,<id>)=<term>-

* detup4
  DEF <term>- where (<id>,<id>,<id>,<id>)=<term>-

* fun2
  DEF λ<id>,<id>.<term>


* c12
  DEF
            1.2:  SETS
            ----
* c12a
  DEF
            1.2.a SETS & POWERSETS
            -----
* sets
  DEF SETS

* setdef
  DEF {<id>:<TYPE>||<TYPE>}

* singleton
  DEF {<element>:<TYPE>}

* carrier
  DEF carrier(<SETS>)

* pred
  DEF pred(<SETS>)

* inset
  DEF <element> ε {<SETS>}

* setconversion
  DEF {<SETS into a set>}

* powerset
  DEF P(<TYPE>)

# set1
  THM >>∀A:TYPE. ∀S:P(A). ∃p:A->U1. S = (A,p) in SETS

# set3
  THM >>∀S:SETS. ∀x:carrier(S). pred(S)(x) <=> x in {S}

# set4
  THM >>∀A:TYPE. ∀S:P(A). ∀x:S. x in A

* c12b
  DEF

            1.2.b NATURAL NUMBERS & NUMBER THEORY
            -----


* leq
  DEF <int>≤<int>

* uneq
  DEF <int>≠<int>

* N
  DEF N

* int_arith
  THM >>∀x,y:int. (x = y in int) | x≠y

* intsub
  THM >>∀i,j:int. i=j in int => j-i = 0 in int

* lessarith
  THM >>∀i,j:int. i<j => 0<j-i

* lessarith1
  THM >>∀i,j,k:int. i≤j & 0≤k => i≤j+k
```

```
* c12c
  DEF
          1.2.c: FINITE SETS & Cardinality
          ------

* nbar
  DEF {1,..,<N>}

* nbar1
  DEF {0,..,<N>}

* finite
  DEF FINITE SETS

* finset
  DEF {<FINITE SETS into a set>}

* finsettype
  THM >>∀A:FINITE SETS. {A} in TYPE

* wf1
  THM >>∀n:N. ∀x:{1,..,n}. x in int

* wf2
  THM >>∀n:N. ∀x:{1,..,n}. x≤n

* wf3
  THM >>∀n:N. ∀x:{1,..,n}. 0 < x

* fineq
  THM >>∀m:N. ∀x:{1,..,(m-1)+1}. x in {1,..,m}

* extend
  THM >>∀k:N. ∀y:{1,..,k}. y in {1,..,k+1}

* extend1
  THM >>∀k:N. ∀x:{1,..,(k-1)+1}. x in {1,..,k+1}

* restrict
  THM >>∀m:N. ∀x:{1,..,m}. x<m => x in {1,..,m-1}

* restrict1
  THM >>∀m:N. ∀x:{1,..,m}. x≠m => x in {1,..,m-1}

* finarith
  THM >>∀m:N. ∀x:{1,..,m}. ( x = m in {1,..,m}) | (x<m)

* finfuntype
  THM >>∀m:N. ∀f:{1,..,m+1}->{1,..,m}. ∀x:{1,..,m+1}. f(x) in int

* hole1
  THM >>∀k:int. 0≤k  => ∀n:N. ∀f:{1,..,k}->{1,..,n}. ∀y:{1,..,n}
        . ∃x:{1,..,k}. f(x)=y in int  |  ∀x:{1,..,k}. f(x)≠y

* pigeon_type
  THM >>∀k:N. ∀f:({1,..,k+1} -> {1,..,k}). ∀i,j:{1,..,k+1}
       . (i<j & f(i) = f(j) in int) in U1

* pigeon
  THM >>∀k:int. 0≤k => ∀f:{1,..,k+1} -> {1,..,k}
       . ∃i,j:{1,..,k+1}
       . i<j & f(i) = f(j) in int

* pigeon_variant
  THM >>∀n:N. ∀h:{0,..,n} -> {1,..,n}
       . ∃k,l:int
       . 0≤k & k<l & l≤n &   h(k) = h(l) in {1,..,n}
```

```
* c13
  DEF
          1.3:     WORDS
          ----
          For convenience strings are viewed as lists over
          the alphabet int.

* c13a
  DEF
          1.3.a: basic definitions
          -----

* symbols
  DEF SYMBOLS

* words
  DEF WORDS

* eps
  DEF ε

* noteps
  DEF <WORDS>≠ε

* concat
  DEF (<WORDS>∘<WORDS>)

* sym
  DEF ⊗<SYMBOLS into WORDS conversion>

* anticons
  DEF <WORDS>*<SYMBOLS>

* rev
  DEF ←<WORDS>

* hd
  DEF hd(<WORDS>)

* tl
  DEF tl(<WORDS>)

* lg
  DEF |<WORDS>|

* iter
  DEF <WORDS>↑<N>

* cutprefix
  DEF <WORDS>[<N>+1..lg]

* select
  DEF <WORDS>(<N>)

* cutsuffix
  DEF <WORDS>[1..<N>]

* range
  DEF <WORDS>[<N>+1..<N>]

* c13b
  DEF

          1.3.b: basic properties of words
          -----

* eps1
  THM >>∀v:WORDS. (ε∘v) = v in WORDS

* eps2
  THM >>∀v:WORDS. (v∘ε) = v in WORDS

* symconcat
  THM >>∀w:WORDS. ∀a:SYMBOLS. (⊗a∘w) = a.w in WORDS

* symcon
  THM >>∀u,v:WORDS. ∀a:SYMBOLS. (a.u∘v) = a.(u∘v) in WORDS

* conassoz
  THM >> ∀u,v,w:WORDS. ((u∘v)∘w) = (u∘(v∘w)) in WORDS

* epsrev
  THM >>←ε = ε in WORDS

* symrev
  THM >>∀a:SYMBOLS. ←⊗a = ⊗a in WORDS
```

```
* consrev
  THM >>∀a:SYMBOLS. ∀u:WORDS. ←a.u = ←u*a in WORDS

* anticonsrev
  THM >>∀w:WORDS. ∀a:SYMBOLS. ←w*a = a.←w in WORDS

* revcon
  THM >>∀u,v:WORDS. ←((u∘v)) = (←v∘←u) in WORDS

* doublerev
  THM >>∀u:WORDS. ←(←u) = u in WORDS

* hdprop
  THM >>∀a:SYMBOLS. ∀u:WORDS. hd(a.u) = a in SYMBOLS

* hd1
  THM >>∀u,v:WORDS. u≠ε => hd((u∘v)) = hd(u) in SYMBOLS

* tlprop
  THM >>∀a:SYMBOLS. ∀u:WORDS. tl(a.u) = u in WORDS

* tl1
  THM >>∀a:SYMBOLS. ∀u,v:WORDS. tl((a.u∘v)) = (u∘v) in WORDS

* tlcon
  THM >>∀u,v:WORDS. u≠ε => tl((u∘v)) = (tl(u)∘v) in WORDS

* lgtype1
  THM >>∀w:WORDS. 0≤|w|

* lgprop
  THM >>∀w:WORDS. w = ε in WORDS <=> |w| = 0 in int

* lgprop1
  THM >>∀w:WORDS. w≠ε <=> 0<|w|

* lgeps
  THM >>|ε| = 0 in int

* lgcons
  THM >>∀a:SYMBOLS. ∀u:WORDS. |a.u| = (|u|+1) in int

* lgsym
  THM >>∀a:SYMBOLS. |⊗a| = 1 in int

* lgconcat
  THM >>∀u,v:WORDS. |(u∘v)| = (|u|+|v|) in int

* lganti
  THM >>∀w:WORDS. ∀a:SYMBOLS. |w*a| = |w|+1 in int

* lgrev
  THM >>∀w:WORDS. |←w| = |w| in int

* lgtl
  THM >>∀w:WORDS. w≠ε => |tl(w)| = (|w|-1) in int

* lgpre
  THM >>∀w:WORDS. ∀k:int. 0≤k & k≤|w|  => |w[k+1..lg]| = |w|-k in int

* lgsel
  THM >>∀w:WORDS. ∀i:int. |⊗w(i)| = 1 in int

* lgsuf
  THM >>∀w:WORDS. ∀l:int. 0≤l => |w[1..l]| = l in int

* lgrange
  THM >>∀w:WORDS. ∀l,r:int. 0≤l & l≤r  => |w[l+1..r]| = r-l in int

* iter1
  THM >>∀w:WORDS. ∀k:int. k<1 => w↑k = ε in WORDS

* iter2
  THM >>∀w:WORDS. ∀k:int. 0<k => w↑k = (w∘w↑k-1) in WORDS

* word1
  THM >>∀l:WORDS. ∀a:SYMBOLS. ∃z:WORDS. ∃b:SYMBOLS. a.l = z*b in WORDS

* noteps1
  THM >>∀w:WORDS. (w = ε in WORDS) | w≠ε

* notepsthm
  THM >>∀w:WORDS. w≠ε <=> ¬w=ε in WORDS
```

```
* preeps
  THM >>∀k:int. ε[k+1..lg] = ε in WORDS

* pre0
  THM >>∀u:WORDS. ∀i:int. 0≤i & i<|u| => u[i+1..lg]≠ε

* pre1
  THM >>∀w:WORDS. w[0+1..lg] = w in WORDS

* pre2
  THM >>∀w:WORDS. ∀k:N. w[k+1+1..lg] = tl(w[k+1..lg]) in WORDS

* pretl
  THM >>∀w:WORDS. ∀k:int. tl(w[k+1..lg]) = tl(w)[k+1..lg] in WORDS

* precons
  THM >>∀a:SYMBOLS. ∀l:WORDS. ∀k:int. 0<k
       => a.l[k+1..lg] = l[k-1+1..lg] in WORDS

* pre3
  THM >>∀w:WORDS. w[|w|+1..lg] = ε in WORDS

* pre4
  THM >>∀w:WORDS. ∀a:SYMBOLS. w*a[|w|+1..lg] = ⊗a in WORDS

* pre5
  THM >>∀u,v:WORDS. ∀i:int. 0≤i & i≤|u|
       => (u∘v)[i+1..lg] = (u[i+1..lg]∘v) in WORDS

# selrev
  THM >>∀w:WORDS. ∀i:{0,..,|w|}. ←w(i) = w(|w|-i) in SYMBOLS

* selcon1
  THM >>∀u,v:WORDS. ∀i:int. 0<i & i≤|u|  => (u∘v)(i) = u(i) in SYMBOLS

# selcon2
  THM >>∀u,v:WORDS. ∀i:int. |u|<i  => (u∘v)(i)= v(i-|u|) in SYMBOLS

* sel1
  THM >>∀w:WORDS. w(1) = hd(w) in SYMBOLS

# sel2
  THM >>∀a:SYMBOLS. ∀w:WORDS. ∀i:N. a.w(i+1) = w(i) in SYMBOLS

# sel3
  THM >>∀w:WORDS. ∀i:{1,..,|w|}. w(i).w[i+1..lg] = w[i-1+1..lg] in WORDS

# sel4
  THM >>∀w:WORDS. ∀i:{1,..,|w|}. (w(i)∘w[i+1..lg])=w[i-1+1..lg] in WORDS

# sel5
  THM >>∀w:WORDS. ∀l,j:N. w[l+1..lg](j) = w(l+j) in SYMBOLS

* sel6
  THM >>∀w:WORDS. ∀a:SYMBOLS. w*a(|w*a|) = a in SYMBOLS

* suf1
  THM >>∀w:WORDS. w[1..0] = ε in WORDS

* suf2
  THM >>∀w:WORDS. ∀l:N. w[1..l+1] = w[1..l]*w(l+1) in WORDS

* sufcon
  THM >>∀u,v:WORDS. ∀k:int. k≤|u|  => (u∘v)[1..k] = u[1..k] in WORDS

* suf3
  THM >>∀w:WORDS. w[1..|w|] = w in WORDS

# suf4
  THM >>∀w:WORDS. ∀l,i:int. i≤l  => w[1..l](i) = w(i) in SYMBOLS

* suf5
  THM >>∀u,v:WORDS. (u∘v)[1..|u|] = u in WORDS

* range1
  THM >>∀w:WORDS. ∀r:int. w[0+1..r] = w[1..r] in WORDS

* range2
  THM >>∀w:WORDS. ∀l:int. 0≤l & l≤|w| => w[l+1..|w|]=w[l+1..lg] in WORDS

# range3
  THM >>∀w:WORDS. ∀l:{1,..,|w|}. ∀r:int. l≤r
       => w[l-1+1..r] = w(l).w[l+1..r] in WORDS
```

```
* range4
  THM >>∀w:WORDS. ∀l,r:int. 0≤l & l<r
       => w[l+1..r] = w[l+1..r-1]*w(r) in WORDS

* range5
  THM >>∀w:WORDS. ∀r:N. w[r+1..r] = ε in WORDS

* range5a
  THM >>∀w:WORDS. ∀l,r,k:int. 0≤l & l≤r & 0≤k
       => (w[l+1..r]∘w[r+1..r+k]) = w[l+1..r+k] in WORDS

* range6
  THM >>∀w:WORDS. ∀l,r,k:int. 0≤l & l≤r & r≤k
       => (w[l+1..r]∘w[r+1..k]) = w[l+1..k] in WORDS

* range7
  THM >>∀w:WORDS. w[0+1..|w|] = w in WORDS

# range8
  THM >>∀w:WORDS. ∀l,r,i:N. l≤r  => w[l+1..r](i) = w(l+i) in SYMBOLS

* range9
  THM >>∀w:WORDS. ∀l,r:int. 0≤l & l≤r
       => (w[1..l]∘w[l+1..r]) = w[1..r] in WORDS

* rangethm
  THM >>∀w:WORDS. ∀l,r:int. 0≤l & l≤r & r≤|w|
       => w = ((w[1..l]∘w[l+1..r])∘w[r+1..lg]) in WORDS

* c15
  DEF
           1.5:     Induction
           ----

* induction_thm
  THM >>∀P:WORDS->U10. (∀v:WORDS. P(v)) <=> (∀u:WORDS. P(←u))

* ind_principle
  THM >>∀P:WORDS->U10
      . P(ε) & ∀b:SYMBOLS. ∀z:WORDS. P(z)=>P(z*b)  =>  ∀w:WORDS. P(w)

* induction_thm1
  THM >>∀P:WORDS->U10.
           ∀i:int. ∀w:WORDS. |w| = i in int => P(w)
       => ∀w:WORDS. P(w)

* ind_principle1
  THM >>∀P:WORDS->U10
      . (P(ε) &
         ∀i:int. 0<i =>       ∀z:WORDS.|z|=(i-1) in int => P(z)
                          => ∀z:WORDS.|z|=i in int      => P(z)
        )
       => ∀w:WORDS. P(w)

# ind_principle2
  THM >>∀P:WORDS->U10
       . ( P(ε) &
           ∀i:int. 0<i =>     ∀v:WORDS.|v|<i         => P(v)
                          => ∀u:WORDS.|u|=i in int => P(u)
         )
         => ∀w:WORDS. P(w)

* trk
  DEF TRK(<A->B>,<B#SYMBOLS->B>)

* trktype
  THM >>∀A,B:TYPE. ∀g:A->B. ∀h:(B#SYMBOLS)->B. TRK(g,h) in (A#WORDS)->B

* trk1
  THM >>∀A,B:TYPE. ∀g:A->B. ∀h:(B#SYMBOLS)->B. ∀x:A
      . TRK(g,h)(x,ε) = g(x) in B

* trk2
  THM >>∀A,B:TYPE.∀g:A->B.∀h:(B#SYMBOLS)->B. ∀x:A. ∀w:WORDS.∀a:SYMBOLS
      . TRK(g,h)(x,w*a) = h(TRK(g,h)(x,w),a) in B

* idtrk
  DEF <B#SYMBOLS->B>*

* trk3
  THM >>∀A,B:TYPE. ∀g:A->B. ∀h:(B#SYMBOLS)->B. ∀x:A. ∀w:WORDS
      . TRK(g,h)(x,w) = h*(g(x),w) in B

* trk4
  THM >>∀B:TYPE. ∀h:(B#SYMBOLS)->B. ∀u,v:WORDS. ∀x:B
      . h*(x,(u∘v)) = h*(h*(x,u),v) in B
```

```
* c2
  DEF
            CHAPTER 2:     FINITE AUTOMATA
            ------------------------------

* c21
  DEF
            2.1:     Basic definitions
            ----

* states
  DEF STATES

* dfa
  DEF DFA

* delstar
  DEF <QxSYMBOLS->Q>*

* delstartype
  THM >>∀Q:STATES. ∀d:({Q}#SYMBOLS) ->{Q}. d* in ({Q}#WORDS) -> {Q}

* delprop
  THM >>∀Q:STATES. ∀d:({Q}#SYMBOLS) ->{Q}. ∀q:{Q}. d*(q,ε) = q in {Q}

* delprop0
  THM >>∀Q:STATES. ∀d:({Q}#SYMBOLS) ->{Q}. ∀q:{Q}. ∀w:WORDS. ∀a:SYMBOLS
        . d*(q,w*a) = d(d*(q,w),a) in {Q}

* delprop1
  THM >>∀Q:STATES. ∀d:({Q}#SYMBOLS)->{Q}. ∀v,w:WORDS. ∀q:{Q}
      . d*(q,(v∘w)) = d*(d*(q,v),w) in {Q}

* accepted
  DEF <WORDS> ε L(<DFA>)

* acceptedtype
  THM >>∀M:DFA. ∀w:WORDS. w ε L(M) in U1

* accept
  DEF L(<DFA>)

* load-all
  ML

* c3
  DEF

            CHAPTER 3:     PROPERTIES OF REGULAR SETS
            ----------------------------------------

* c31
  DEF
            3.1: PUMPING LEMMA
            ----


* pumping_type
  THM >>∀M:DFA. ∀n:int. ∀z:WORDS. ∀u,v,w:WORDS
       . ( z=((u∘v)∘w) in WORDS & |(u∘v)|≤n & 0 < |v|
           & ∀k:int. ((u∘(v↑k))∘w) ε L(M)
         ) in U1

* pumping
  THM >>∀M:DFA. ∃n:int. ∀z:WORDS. z ε L(M) & n≤|z|  => ∃u,v,w:WORDS
        . z=((u∘v)∘w) in WORDS & |(u∘v)|≤n & 0 < |v|
        & ∀k:int. ((u∘(v↑k))∘w) ε L(M)

* pumping_demo
  EVAL

* end
  DEF
           END AUTOMATA THEORY
```

Appendix B: Extensions of the PRL object language

Description of the ML-functions and tactics defined in the files

supply.ml	(general extensions)
tactics.ml	
logic.ml	(logic notation)
tupling.ml	(pairing, triples, quadruples)
sets.ml	(something about sets for the purpose of using powersets)
finset.ml	(finite sets {1,..,n} etc.)
words_term.ml	(words (= strings) and operators on words)
words_rules_1.ml	
words_rules_compute.ml	
words_tactics.ml	
words_recursion.ml	(recursive function definitions on words - written separately)
dfa.ml	(deterministic finite automata)

Definition instatiations refer to the PRL library "**automata**"

```
-----------------------------------------------------------------------------+
 Supply.ml:      ML-functions for general use in PRL                         |
-----------------------------------------------------------------------------+


, **************** LIST OF USED PREDEFINED FUNCTIONS  ***************************
                                                                               *
     map           :  (* -> **)#(*list) -> (**list)                            *
     ^             :  tok -> tok -> tok               (infix concat)           *
     three_match   :  (term#term#tok list) -> term list                        *
                                                                               *
******************* END LIST OF PREDEFINED FUNCTIONS **************************%


basic term definitions
   INT     : term                 - short for "make_int_term"
   NIL     : term                 - short for "make_nil_term"
   VOID    : term                 - short for "make_void_term"
   mvar    : tok -> term          - short for "make_var_term"
   imp     : term                 - dummy term
   andl    : term                 - dummy term
   andr    : term                 - dummy term
   zero    : term                 - PRL "0"
   one     : term                 - PRL "1"
   U1      : term                 - PRL "U1"
   U2      : term                 - PRL "U2"


elpfunctions
  cat             : tok -> int -> tok                concats token and integer to a token
  is_member       : * -> *list -> bool               checks if an element is in a list
  ids             : declaration_list -> tok_list     creates list of identifiers of the dec_list
  cut_from        : int -> * list -> * list          i [a1;..an] -> [a1,..ai]
  cut_first       : int -> * list -> * list          i [a1;..an] -> [ai,..an]

  is_alpha_convertible : term -> term -> bool

  list            : int -> * -> * list               n a  -> [a;a;a;..]
  intlist         : int -> int -> int list           n x ->  [x;x+1;...x+n-1]

  first_exp       : proof -> term                    if conclusion is a=b in T return a
  ordered_equality: int -> proof -> (term#term#term)
                    if conclusion is a=b in T return a,b,T for no = 1  b,a,T otherwise

  new             : tok -> proof -> tok
                    returns a new identifier not used in the proof so far which has the given token as prefix.
                    does not check simultaneous calls (should be faster than new_id)
                    In that case use different tokens to distinguish

 YPE INFORMATION FUNCTIONS
  try to find the type of a given expression using the informations given in a proof

  hyp_info    : tok -> proof -> int#term
                returns hypothesis number and type if var is mentioned in declaration list - fails otherwise

  typed       : term -> declaration -> term;
                returns type of an expression if mentioned in declaration - fails otherwise

  typeof      : term -> proof -> (term -> proof -> term) -> term
  type_of     : term -> proof -> term
                typeof returns the type of an expression using special_type_information first, information from
                the hypotheses of the proof and information from the structure of the expression.
                 - special_type_information must fail if it does not succeed
                 - default is (\exp.\proof. fail) as in type_of
```

ATCHING

match_subterms : term -> term -> proof -> term list

matches a term s against subterms of a given term t and returns an instantiation list for variables in t. Tries to find additional information on instances in the given proof if instance_list is incompe.

EXAMPLE:

```
t = (x1:A1 -> A2 -> ( (x3:A3 -> B & B') # (y3:C3 ->C4) )
s = B'[a1,a3/x1,x3]
```

returns a1.imp.andl.a3.andr

Needs a lot of helpfunctions and the following lists

match_list	[...(xi,ai)..]	generated by match, maybe unordered or incomplete
elim_list	[...(xi,Ai)..]	generated during decomposition of t, ordered during build (a helpfunction) complete from the i-th element
inst_list	ai.ai+1.....	instances for the xi (resp. imp, andl, andr) found in match_list or otherwise ordered and complete from the i-th element

```
%------------------------------------------------------------------------------------------+
| tactics.ml:      Tactics for general use in PRL                                          |
+------------------------------------------------------------------------------------------+

% **************************** LIST OF USED PREDEFINED TACTICS  ****************************
'                                                                                          *
' From :>prl>ml>tactics-1 (-2/ -3/ -4)                                                     *
'       Elim                    : int -> tactic                                            *
'       RepeatFor               : int -> tactic -> tactic                                  *
'       ComputeConclUsing       : (term -> term) -> tactic                                 *
'       NormalizeConcl          : tactic                                                   *
'       NormalizeHyp            : int -> tactic                                            *
'       NormalizeHyps           : int list -> tactic                                       *
'       TopLevelComputeConcl    : tactic                                                   *
'       TopLevelComputeHyp      : int -> tactic                                            *
'       SubstFor                : term -> tactic                                           *
'       Cases                   : term list -> tactic                                      *
'       ThinToEnd               : int -> tactic                                            *
'       RepeatAndElim           : int -> tactic                                            *
                                                                                           *
**************************** END LIST OF PREDEFINED TACTICS *****************************%

lementary tactics & Rule-tactics
--------------------------------
  Intro                  = tactic                   PRL-rule "intro"
  Set_elim               = int -> int -> tactic     PRL-rule "elim" hyp level      for sets

  Hyp                    = int -> tactic            PRL-rule "hyp"
  Last_hyp               = tactic                   use last hypothesis
  Last_1_hyp             = tactic                   use hypothesis before the last one
  Last_2_hyp             = tactic

  CUMULATIVITY           = int -> tactic            PRL-rule "cumulativity" level
  Cumulativity           = tactic                   PRL-rule "cumulativity at U1"
  Thinning               = int_list -> tactic       PRL-rule "thinning"
  ThinLast               = int -> tactic            thinning of the last n hypotheses
  Thin_last              = tactic                   thinning of the last hypothesis

  equal                  = tactic                   PRL-Rule "equality"
  Equal                  = tactic                   TRY equal
  Arith                  = tactic                   PRL-rule "arith at U1"
  Decision               = tactic                   equal ORELSE (COMPLETE Arith)

  Lemma                  = tok -> tactic            PRL-rule "lemma" name
  Seq                    = term list -> tactic      PRL-rule "seq"
  Extensionality         = tactic                   PRL-rule "extensionality"

  FAILTAC                = tactic                   fails always

  Int_minus              = tactic                   rule integer_equality_minus
  Int_add                = tactic                   rule integer_equality_addition
  Int_sub                = tactic                   rule integer_equality_subtraction
  Int_mult               = tactic                   rule integer_equality_multiplication
  Int_mod                = tactic                   rule integer_equality_modulo
  Int_number             = tactic                   rule integer_equality_natural_number

  Ui_equality            = tactic                   rule equal_equality
  UiUjtac                = tactic                   rule universe_equality
  ProdUitac              = tactic                   rule product_equality_independent
  FuncUitac              = tactic                   rule function_equality_independent

  Int_computation        = tok -> int -> tactic     rule integer_computation kind no
  Inteq_computation      = int -> bool -> tactic    rule int_eq_computation no bool

  List_computation       = int -> tactic            rule list_computation no

  Int_elim               = int -> tactic            PRL-rule "elim" hyp    for integers  (induction)

  Direct_computation     = term -> tactics          rule direct_computation term
  Hyp_computation        = int -> term ->tactic     rule direct_computation_hyp hyp term

  Trivial_types          = tactic                   a collection of all the Intro-rules which don't leave subgoal
```

omputation tactics (TO BE COMPLETED)

```
H >> redex                              Compute howoften
  >> contractum

H >> redex in T                         Compute_term howoften
  >> contractum in T

H >> redex = t in T                     Compute_term howoften
  >> contractum = t in T

H >> t = redex in T                     Compute_snd_term howoften
  >> t = contractum in T

H >> t in redex                         Compute_type howoften
  >> t in contractum

H >> y:redex -> T                       Compute_function_left howoften
  >> y:contractum -> T

H >> y:T -> redex                       Compute_function_right howoften
  >> y:T -> contractum

H >> t in y:T -> redex                  Compute_equal_function_right howoften
  >> t in y:T -> contractum

H >> t in y:redex -> T                  Compute_equal_function_left howoften
  >> t in y:contractum -> T
```

The same for products, sets, unions, applications, lists (some day.)

The same for hypotheses :

Hyp_compute hyp howoften
Hyp_compute_term hyp howoften
Hyp_compute_snd_term hyp howoften
Hyp_compute_type hyp howoften

(Definition instantiations will be destroyed - so use these only within completion tactics)

lementary tacticals

THENMEMBER = tactic -> tactic -> tactic   apply tac2 to all membership subgoals of tac1

REPEATL = (tactic -> tactic) -> tactic list -> tactic
repeat application of the tactical to all tactics in the tactic list

ypothesis elimination

elim_using : tactic -> term list -> int -> tactic
eliminate_using : term list -> int -> tactic

Given an instantiation_list of the form [a1;a2;...an] where ai might be either instance term or one of the special terms "imp", "andl" or "andr"
elim_using eliminates the named hypothesis hyp corresponding to the ai's
- nulltac will be used if inst_list is empty
- subgoals that the ai are of the right type are left over

Example:

```
      hypothesis hyp be      "x1:T1 -> x2:T2 -> (x3:T3 -> T4 & T4') & (y3:T3'-> T4'')
      and  inst_list   =     [a;imp;andl;b;andr]

      Result  H >> T by elim_using .....
                 >> a in T1
                 >> T2
                 >> b in T3
                 T4'[a,b/x1,x3] >> T
```

xtended hypothesis tactic

```
  Direct_hypotheses              = tactic                           try all hypotheses
  Hyp_elim_on                    = term list -> int -> tactic       same as eliminate_using
  HYP                            = int -> tactic                      try if goal follows from named hyp via
                                                                      function or product elimination
                                                                      (uses matching)
  Try_elimination_in_hypotheses  = tactic                             try if goal follows from hypotheses via
                                                                      function or product elimination
                                                                      (uses matching)
  HYPOTHESIS                     = tactic
                                   tries all kinds of manipulations with hypotheses which end up without
                                   leaving subgoals. This includes decision procedures and elimination
                                   within hypotheses if they succeed.
```

embership tactic

```
  membertac      : tactic -> tactic;
  membership     : tactic
                   Intended to solve all kind of membership ("a in T") problems - particularly wellformedness.
                   For increasing speed and specially defined types a tactic "special_tac", which will be applied
                   first, may be used first. special_tac should fail if it is not applicable.
                   - Default is FAILTAC which leads to membership
                   - "SIDE-effect": proofs "x = y in T" which can be solved by introduction steps
                     (except set_equality and quotient_equality and formation) are also covered by membership
                     sometimes somewhat unpredictable (needs improvement but under a different name)
```

HEOREM APPLICATION   (including elimination of ∀ and & if necessary *)

```
  THEOREM1         : tok -> term list -> tactic
  THEOREM          : tok -> tactic

       the named theorem be (as an example)

             >>  ∀x1:T1. ∀x2:T2. (T3 => (( ∀x4:T4. T5) & (S4 => S5)) ) & S3

       THEOREM1 needs a given instantiation list and produces

             H >> T by THEOREM1 name [t1;t2;andl;imp;andl;t4]
                     >> t1 in T1
                     >> t2 in T2[t1/x1]
                     >> T3[t1,t2/x1,x2]
                     >> t4 in T4[t1,t2/x1,x2]
                     T5[t1,t2,t4/x1,x2,x4] >> T


      THEOREM  tries to find the instantiation list itself but for this the goal T
               It has to match one of the theorems subterms (e.g. T5[t1,t2,t4/x1,x2,x4])

             H >> T by THEOREM name
                     >> t1 in T1
                     >> t2 in T2[t1/x1]
                     >> T3[t1,t2/x1,x2]
                     >> t4 in T4[t1,t2/x1,x2]

      No attempts will be made in the above THEOREM rules to solve one ove the subgoals.
      A combination of THEOREM/THEOREM1 with a membership tactic will do that.

  theorem          : tok -> term list -> tactic
  thm              : tok -> tactic
```

* ∀ and & are defined as in the logic section - the implementation of THEOREM however is independent of this file.

```
;--------------------------------------------------------------------------------------------+
  logic.ml:     PRL-extensions for logic definitions                                        |
--------------------------------------------------------------------------------------------%


 . Constructors, Destructors & Predicates
 ----------------------------------------

 a:  Constructors
 -----------------
  make_and_term         = term -> term -> term                          : p q             into  "p&q"
  make_or_term          = term -> term -> term                          : p q             into  p|q
  make_imp_term         = term -> term -> term                          : p q             into  "p=>q"
  make_equiv_term       = term -> term -> term                          : p q             into  "p<=>q"
  make_false_term/FALSE = term                                          :                       "false"
  make_not_term         = term -> term                                  : p               into  "¬p"
  make_all_term         = term -> term -> term -> term                  : x type term     into  "∀x:type. term"
  make_all2_term        = term -> term -> term -> term -> term          : x y type term   into  "∀x,y:type. term"
  make_all3_term        = term -> term -> term -> term -> term -> term : x y z type term into  "∀x,y,z:type. term
  make_some_term        = term -> term -> term -> term                  : x type term     into  "∃x:type. term"
  make_some2_term       = term -> term -> term -> term -> term          : x y type term   into  "∃x,y:type. term"
  make_some3_term       = term -> term -> term -> term -> term -> term : x y z type term into  "∃x,y,z:type. term

 p,q,type,term  are placeholders for terms in Ui,  x,y,z for var_terms


 b:  Destructors
 ----------------
  split the constructions named above into their parts. The result is a product term.
  e.g. destruct_all2    = term -> (term#term#term#term)                 : "∀x,y:type. term" into  x,y,type,term

  The names of these destructors are obvious.


 c: Predicates
 ---------------
  They detect if a term represents a particular product. Due to simulation of my types by
  ordinary PRL-types not all constructs are disjoint.

       not is a subconstruct of  imp    (i.e. if is_not_term t  is true so is  is_imp_term t)
       equiv                   and
       all3   of all2           of  all
       some3  of some2          of  some

  One has to remember this if one does case analysis using logic-terms.
```

```
I. Rules
--------

Ia: Formation                                         (-->>> lintro isn't written yet)
--------------
  H >> A&B in Ui by lintro                                    and_equality
     >> A in Ui
     >> B in Ui

  H >> A|B in Ui by lintro                                    or_equality
     >> A in Ui
     >> B in Ui

  H >> A=>B in Ui by lintro                                   imp_equality
     >> A in Ui
     >> B in Ui

  H >> A<=>B in Ui by lintro                                  equiv_equality
     >> A in Ui
     >> B in Ui

  H >> false in Ui by lintro                                  false_equality

  H >> ¬A in Ui by lintro                                     not_equality
     >> A in Ui


  H >> (∀x:A.B) in Ui by lintro                               all_equality
     >> A in Ui
     x:A >> B in Ui

  H >> (∀x1:A1....∀xn:An.B) in Ui by lintro  [n]              repeat_all_equality
     >> A1 in Ui                                              repeat_all_equality_for n
     x1:A1 >> A2 in Ui
       .
     x1:A1,..,xn:An >> B in Ui


  H >> (∃x:A.B) in Ui by lintro                               some_equality
     >> A in Ui
     x:A >> B in Ui

  H >> (∃x1:A1....∃xn:An.B) in Ui by lintro  [n]              repeat_some_equality
     >> A1 in Ui                                              repeat_some_equality_for n
     x1:A1 >> A2 in Ui
       .
     x1:A1,..,xn:An >> B in Ui
```

Ib: Introduction

```
H >> A&B  by lintro                                      and_equality
   >> A
   >> B

H >> A|B by lintro left i                                or_intro_left level
   >> A
   >> B in Ui

H >> A|B by lintro right i                               or_intro_right level
   >> B
   >> A in Ui

H >> A=>B  by lintro i                                   imp_intro level
   A >> B
   >> A in Ui

H >> A<=>B  by lintro i                                  equiv_intro level
  A >> B
  >> A in Ui
  B >> A
  >> B in Ui

H >> false by lintro                                     false_intro (H must be contradictory)

H >> ¬A by lintro i                                      not_intro level
   A >> FALSE
   >> A in Ui

H >> ∀x:A.B by lintro i                                  all_intro level
   x:A >> B
   >> A in Ui


H >> ∀x1:A1....∀xn:An.B  by lintro i [n]                 repeat_all_intro level
   x1:A1,..,xn:An >> B                                   repeat_all_intro_for n level
   x1:A1,..xn-1:An-1 >> An in Ui
    .
   >> A1 in Ui


H >> ∃x:A.B by lintro i a                                some_intro level a
   >> a in A
   >> B[a/x]
   x:A >> B in Ui

H >> ∃x1:A1....∃xn:An.B by lintro i a1..an [n]           repeat_some_intro level alist
   >> A1 in Ui
   x1:A1 >> A2 in Ui
    .
   x1:A1,..,xn:An >> B in Ui
   >>a1 in A1
    .
   >>an in An[a1,..,an-1/x1,..,xn-1]
   >>B[a1,..,an/x1,..xn ]
```

:Ic: Elimination

```
-----------------
  H, A1&A2..&An,H' >> T                                          and_elim hyp
    H, H', A1, A2,.., An >> T

  H, A1|A2|..|An, H' >> T                                        or_elim hyp
     A1 >> T
     A2 >> T
      .
     An >> T

  H, A=>B, H' >> T                                               imp_elim hyp
    >> A
    B >> T

  H, A<=>B, H' >> T                                              equiv_elim_left hyp
    >> A
    B >> T

  H, A<=>B, H' >> T                                              equiv_elim_right hyp
    >> B
    A >> T

  H, FALSE, H' >> T                                              false_elim hyp

  H, ¬A, H' >> T                                                 not_elim hyp
    H, H'>> A

  H, ∀x1:A1....∀xn:An.B, H' >> T                                 all_elim_on inst_list hyp
    >> a1 in A1
     .
    >> an in An[a1,..,an-1/x1,..xn-1]                                (inst_list = [a1;..;an])
    B[a1,..,an/x1,..xn] >> T

  H, ∃x1:A1....∃xn:An.B, H' >> T                                 some_elim hyp
    H, H', x1:A1,.., xn:An, B >> T                               some_elim_for n hyp
```

[d: Computation

```
----------------
```

lo computation for logic expressions

II. Tactics

H >> ∀x:A.B by **AllIntro** j AUjtac
x:A >> B

H >> ∀x1:A1...∀xn:An.B by **RepeatAllIntro** j AiUjtacs
x1:A1,.., xn:An >> B

H >> ∃y:A.B by **SomeIntro** j a atac BUjtac
>> B[a/y]

H >> ∃y1:A1... ∃yn:An.B by **RepeatSomeIntro** j ailist aitacs AiUjtacs BUjtac
>> B[a1,..,an/y1,..yn]

H >> A => B by **ImpIntro** j AUjtac
A >> B

H >> A <=> B by **EquivIntro** j AUjtac BUjtac
A >> B
B >> A

aitacs must prove ">> ai in Ai[a1,..ai-1/x1,..xi-1]"
AiUjtacs ">> Ai in Uj"
BUjtac ">> B in Uj

Using IDTAC instead leaves the corresponding subgoals open

Short forms:
level j = 1
- **All_intro** AU1tac
- **Repeat_all_intro** AiU1tacs
- **Some_intro** a atac BU1tac
- **Repeat_some_intro** ailist aitacs AiU1tacs BU1tac
- **Imp_intro** AU1tac
- **Equiv_intro** AU1tac BU1tac

level j = 1 and AU1tac = membership (+BU1tac = membership)
- **Allintro**
- **Impintro**
- **Equivintro**

to be written:
Repeat_and_equality
Repeat_and_equality_for n
Repeat_and_intro
Repeat_and_intro_for n

11-introduction for standard PRL-types

H >> ∀x:int.T by **int_all_intro**
x:**int** >> T

H >> ∀x:int.T by **int_induction**
x:**int**, k:int, k<0, indhyp: T[k+1/x] >> T [k/x]
x:**int** >> T[0/x]
x:**int**, k:int, 0<k, indhyp: T[k-1/x] >> T [k/x]

to be written
void_all_intro
atom_all_intro

```
;-----------------------------------------------------------------------------------+
  tupling.ml:     PRL-extensions for tupling, multifunctions etc.                   |
;-----------------------------------------------------------------------------------%

. Constructors, Destructors & Predicates
---------------------------------------

a:  Constructors
----------------
  make_tup2_term          = term -> term -> term                         a b         into  "(a,b)"
  make_tup3_term          = term -> term -> term -> term                 a b c       into  "(a,b,c)"
  make_tup4_term          = term -> term -> term -> term -> term         a b c d     into  "(a,b,c,d)"
  make_where2_term        = term -> term -> term -> term                 t a b t'    into  "t - where (a,b) = t'"
  make_where3_term        = term -> term -> term -> term -> term         t a b c t'  into  "t - where (a,b,c) = t'"
  make_where4_term        = term -> term -> term -> term -> term -> term
                                                                 t a b c d t'  into  "t - where (a,b,c,d) = t'"
  make_id_function_term/id = term                                                          "\x.x"
  make_lambda2_term       = term -> term -> term -> term                 x y t       into   "λx,y.t"
  make_apply2_term        = term -> term -> term -> term                 f a b       into   "f(a,b)"

 additional constructors
  make_equal_int_term     = term list -> term                            [x;y;..]    into   "x=y..in int"

Destructors and predicates as usual


I. Rules
--------

Ia:  Formation
--------------
     ----  No formation rules for tupling

Ib: Introduction
----------------
  H >> (a1,a2,..,an) in x1:A1#x2:A2#...#An                       tup_equality level
     >> a1 in A1                                                 tup_equality_for n level
     >> a2 in A2[a1/x1]                                          tup2_equality level
      .                                                          tup3_equality level
     >> an in An [a1,..,an-1/ x1,..,xn-1]                        tup4_equality level
     x1:A1,..,xn-1:An-1 >> An in Ui                                (if xj = `NIL` (independent) then the subgoal
      .                                                            "x1:A1,..xj:Aj >> Aj+1 in Ui" does not appear)
     x1:A1 >> A2 in Ui

  H >> t - where (a,b) = x - in T                                where2_equality using
     >> x in y:A#B
     a1:A, b1:B[a1/y], x=(a1,b1) in y:A#B >> t[a1,b1/a,b] in T

  H >> t - where (a,b,c) = x - in T                              where3_equality using level
     >> x in y1:A1#y2:A2#A3
     y1:A1 >> y2:A2#A3 in Ui                                                   (missing if y1 = `NIL`)
      a1:A, b1:A2[a1/y1], c1:A3[a1,b1/y1,y2], x=(a1,b1,c1) in y1:A1#y2:A2#A3
     >> t[a1,b1,c1/a,b,c] in T

  H >> t - where (a,b,c,d) = x - in T                            where4_equality using level
     >> x in y1:A1#y2:A2#y3:A3#A4
     y1:A1,y2:A2 >> y3:A3#A4 in Ui                                             (missing if y2 = `NIL`)
     y1:A1 >> y2:A2#y3:A3#A4 in Ui                                             (missing if y1 = `NIL`)
      a1:A, b1:A2[a1/y1], c1:A3[a1,b1/y1,y2], d1:A4[a1,b1,c1/y1,y2,y3]
      x=(a1,b1,c1,d1) in y1:A1#y2:A2#y3:A3#A4
     >> t[a1,b1,c1,d1/a,b,c,d] in T

  H >> λa,b.t in y:(x:A#B) -> C                                  lambda2_equality level
     a:A, b:B[a/x] >> t in C
     >> x:A#B in Ui

  H >> f(a,b) in C                                               apply2_equality using level
     >> f in y:(x:A#B) -> C
     >> a in A
     >> b in B[a/x]
     x:A >> B in Ui
```

Ic: Elimination

Id: Computation (TO BE WRITTEN)

H >> t - where (a1,..,an) = (b1,..,bn) - = t' in T by treduce 1 >> t[b1,..,bn/a1,..,an] = t' in T	**tup_reduce** where
H >> (λx,y.t)(a,b) = t' in T by treduce 1 >> t[a,b/x,y] = t' in T	**lambda2_reduce** where
H >> t - where (a1,..,an) = (b1,..,bn) by tcompute >> t[b1,..,bn/a1,..,an]	**tup_compute**
H >> (λx,y.t)(a,b) by tcompute >> t[a,b/x,y]	**lambda2_compute**
H, t - where (a1,..,an) = (b1,..,bn) - = t' in T, **H' >>** T' H, t[b1,..,bn/a1,..,an] = t' in T, H' >> T'	**tup_reducehyp** i 1
H, t - where (a1,..,an) = (b1,..,bn) - , H' >> T' H, t[b1,..,bn/a1,..,an], H' >> T'	**tup_computehyp** i
H, (λx,y.t)(a,b) = t' in T, **H' >>** T' H, t[a,b/x,y] = t' in T, H' >> T'	**lambda2_reducehyp** i 1
H, (λx,y.t)(a,b), H' >> T' H, t[a,b/x,y], H' >> T'	**lambda2_computehyp** i

II. Tactics

COMPUTATION

H >> f(x,redex) [= t] in T >> f(x,contractum) [= t] in T	**Compute_apply2_arg2**
H >> t = f(x,redex) in T >> t = f(x,contractum) in T	**Compute_snd_apply2_arg2**
H >> f(redex,w) [= t] in T >> f(contractum,w) [= t] in T	**Compute_apply2_arg1**
H >> t = f(redex,w) in T >> t = f(contractum,w) in T	**Compute_snd_apply2_arg1**
H >> redex(x,w) [= t] in T >> contractum(x,w) [= t] in T	**Compute_apply2_fun**
H >> t = redex(x,w) in T >> t = contractum(x,w) in T	**Compute_snd_apply2_fun**

```
;----------------------------------------------------------------------------------------+
       sets.ml                          PRL_extensions for the type SETS                  |
-----------------------------------------------------------------------------------------%
```

. Term constructors, predicates and destructors

a: Constructors

```
  make_SETS_term/SETS    = term                                             "SETS"
  make_carrier_term      = term -> term                    S       into  "carrier(S)"
  make_pred_term         = term -> term                    S       into  "pred(S)"
  make_setdef_term       = term -> term -> term -> term    x A B   into  "{x:A||B}"
  make_singleton_term    = term -> term -> term            x A     into  "{x:A}"
  make_inset_term        = term -> term -> term            x S     into  "x ε S"
  make_set_conversion    = term -> term                    S       into  "{S}"
  make_powerset_term     = term -> term                    A       into  "P(A)"
```

estructors and predicates as usual

I. RULES   (not all of these tactics are written yet)

```
   helptactic
     SETS_spread    = tactic
```

Ia: Formation

```
  H >> SETS in Ui by Sintro        (i>1)          SETS_equality              / SETSeq

  H >> {S} in Ui by Sintro                        SETS_equality_conversion / Sconversion
     >> S in SETS

  H >> P(A) in Ui by Sintro        (i>1)          SETS_equality_power        / Spower
     >> A in TYPE

  H >> x ε {S} in Ui by Sintro                    SETS_equality_inset        / Sinset
     >> S in SETS
     >> x in carrier(S)
```

Ib: Introduction

```
  H >> (X,P) in SETS by Sintro                    SETS_equality_pair         / Spair
     >> X in TYPE
     >> P:X -> U1

  H >> S in P(A) by Sintro                        SETS_power_element         / Spowerel
     >> S in SETS
     S in SETS >> carrier(S) = A in U1

  H >> x in {S} by Sintro                         SETS_conversion_element  / Sconvel
     >> x in carrier (S)
     >> pred(S)(x)

  H >> carrier(S) in Ui by Sintro                 SETS_equality_carrier      / Scarrier
     >> S in SETS

  H >> pred(S) in carrier(S)->U1 by Sintro        SETS_equality_pred         / Spred
     >> S in SETS
```

Ic: Elimination

```
  H, S:SETS, H' >> T                                         SETS_elim hyp
     carrier:TYPE, pred:carrier->U1, S = (carrier,pred) in SETS
     >> T[(carrier,pred)/S]

  H, x:{S}, H' >> T                                          conversion_elim hyp
     x in carrier(S), x ε {S} >> T

  H, S:P(A), H' >> T                                         powerset_elim hyp
     >> A in TYPE
     pred:A->U1, S = (A,pred) in SETS >> T[(A,pred)/ S]
```

The last two should include reductions of the hypotheses if possible)

Id: Computation

II: Miscellaneous functions and tactics
----

sets_all_intro = tactic   all-introduction for SETS

```
;----------------------------------------------------------------------------------------+
  finset.ml:     PRL-extensions for natural numbers/ arithmetic/ finite sets etc.        |
-----------------------------------------------------------------------------------------%

. Constructors, Destructors & Predicates
----------------------------------------

a: Constructors
---------------
 make_leq_term                  : term -> term -> term       i j  into "i≤j"
 make_uneq_term                 : term -> term -> term       i j  into "i≠j"
 make_nat_term/nat              : term                                 "N"
 make_nbar_term                 : term -> term               j     into "{1,..,j}"
 make_nbar1_term                : term -> term               j     into "{0,..,j}"
 make_finite_term/finite_sets   : term                                 "FINITE SETS"
 make_finset_term               : term -> term               A     into "{A}"

Destructors and predicates as usual


I. Rules                                                     (nearly all of these rules aren't written yet)
--------

Ia: Formation
-------------
  H >> i≤j in Ui by fintro                                   leq_intro
     >> i in int
     >> j in int

  H >> i≠j in Ui by fintro                                   uneq_intro
     >> i in int
     >> j in int

  H >> N in Ui by fintro                                     nat_intro

  H >> {1,..,n} in Ui by fintro                              nbar_intro
     >> n in int

  H >> {0,..,n} in Ui by fintro                              nbar1_intro
     >> n in int

  H >> FINITE SETS in Ui by fintro                           finset_intro

  H >> {A} in Ui by fintro                                   finset_equality_conversion
     >> A in FINITE SETS

Ib: Introduction
----------------

Ic: Elimination
---------------

Id: Computation
---------------
 these should include rules like (see theorems in chapter 12b/12c)

  H >> x in {1,..,k+1}
     >> x in {1,..k}

  H >> x in {1,..,(k+1)-1}
     >> x in {1,..k}

  H >> x in {1,..,k-1}
     >> x in {1,..,k}
     >> x ≠ k

  H >> 0< j-i
     >> i < j
```

```
%-----------------------------------------------------------------------------------------+
       words_term.ml                   PRL_extensions for strings                        |
+-----------------------------------------------------------------------------------------%


1. Term constructors, predicates and destructors
-----------------------------------------------

a:  Constructors
----------------

1. PRL-TYPES    corresponding to the definitions in the library "automata"

  make_symbols_term/symbols     = term                            :               "SYMBOLS"
  make_words_term/words         = term                            :               "WORDS"
  make_noteps_term              = term -> term                    : w       into  "w≠ε"
  make_eps_term/eps             = term                            :               "ε"
  make_cons_term                = term -> term -> term            : a l     into  a.l
  make_concat_term              = term -> term -> term            : u v     into  "(u∘v)"
  make_sym_term                 = term -> term                    : a       into  "⊗a"
  make_anticons_term            = term -> term -> term            : w a     into  "w*a"
  make_rev_term                 = term -> term                    : w       into  "←w"
  make_iter_term                = term -> term -> term            : w i     into  "w↑i"
  make_hd_term                  = term -> term                    : w       into  "hd(w)"
  make_tl_term                  = term -> term                    : w       into  "tl(w)"
  make_lg_term                  = term -> term                    : w       into  "|w|"
  make_cutprefix_term           = term -> term -> term            : w i     into  "w[i+1..lg]"
  make_select_term              = term -> term -> term            : w i     into  "w(i)"
  make_cutsuffix_term           = term -> term -> term            : w i     into  "w[1..i]"
  make_range_term               = term -> term -> term -> term    : w i j   into  "w[i+1..j]"

2. additional constructors which are useful in rules and tactics

  make_equal_word_term          = term list -> term       : [w1;..;wn]  into  "w1=..=wn in WORDS"
  make_equal_symbols_term       = term list -> term       : [w1;..;wn]  into  "w1=..=wn in SYMBOLS"

 u,v,w,l are placeholders for terms of type WORDS,  a  for SYMBOLS,  i,j for int.


b:  Destructors
---------------
  split the constructions named above into their parts. The result is a product term.
  e.g.  destruct_concat           = term -> (term#term)     : "(u∘v)" into  u,v.

  The names of these destructors are obvious.


c:  Predicates
--------------
  They detect if a term represents a particular product. Due to simulation of my types by
  ordinary PRL-types not all constructs are disjoint.

  anticons is a subconstruct of concat (i.e. if is_anticons_term t is true so is is_concat_term t)
  sym                           cons
  select                        hd
  range                         cutsuffix

 One has to remember this if one does case analysis using word-terms.
```

```
;----------------------------------------------------------------------------------------+
       words_rules_1.ml                             PRL_extensions for strings           |
.----------------------------------------------------------------------------------------%
```

I. RULES

Ia: Formation

```
  H >> SYMBOLS in Ui          by wintro                 symbol_equality          / sequal

  H >> WORDS in Ui            by wintro                 word_equality            / wequal

  H >> w≠ε in Ui              by wintro                 noteps_equality          / noteps
     >> w in WORDS
```

Ib: Introduction

```
  H >> ε in WORDS             by wintro                 word_equality_eps         / weps

  H >> a.v in WORDS           by wintro                 word_equality_cons        / wcons
     >> a in SYMBOLS
     >> v in WORDS

  H >> u•v in WORDS           by wintro                 word_equality_concat      / wconcat
     >> u in WORDS
     >> v in WORDS

  H >> ⊗a in WORDS            by wintro                 word_equality_symbols     / wsym
     >> a in SYMBOLS

  H >> u*a in WORDS           by wintro                 word_equality_anticons    / wanti
     >> u in WORDS
     >> a in SYMBOLS

  H >> ←w in WORDS            by wintro                 word_equality_rev         / wrev
     >> w in WORDS

  H >> w↑i in WORDS           by wintro                 word_equality_iter        / witer
     >> i in int
     >> w in WORDS

  H >> hd(w) in SYMBOLS       by wintro                 word_equality_hd          / whd
     >> w in WORDS

  H >> tl(w) in WORDS         by wintro                 word_equality_tl          / wtl
     >> w in WORDS

  H >> |w| in int             by wintro                 word_equality_lg          / wlg
     >> w in WORDS

  H >> w[j+1..lg] in WORDS    by wintro                 word_equality_cutprefix   / wpre
     >> j in int
     >> w in WORDS

  H >> w(i) in SYMBOLS        by wintro                 word_equality_select      / wselect
     >> i in int
     >> w in  WORDS

  H >> w[1..i] in WORDS       by wintro                 word_equality_cutsuffix   / wsuf
     >> i in int
     >> w in WORDS

  H >> w[l+1..r] in WORDS     by wintro                 word_equality_range       / wrange
     >> l in int
     >> r in int
     >> w in WORDS
```

Ic: Elimination (T uses a theorem)

There is no SYMBOL elimination

```
H, w:WORDS, H'  >> T                                          word_elim hyp
  >> T[ε/w]
  hd:SYMBOLS, tl:WORDS, tl_hyp:T[tl/w] >> T[hd.tl/w]

H, w:WORDS, H'  >> T                                          word_elim_tail  hyp
  u:WORDS >> T[u/w] in U10
  >> T[ε/w]
  a:SYMBOLS, v:WORDS, T[v/w] >> T[v*a/w]

H, w:WORDS, H'  >> T                                          word_elim_lg  hyp
  u:WORDS >> T[u/w] in U10
  >> T[ε/w]
   i:int, 0<i, (∀v:WORDS. |v|=i-1 in int => T[v/w]), u:WORDS, |u|=i in int
  >> T[u/w]
```

Additional Helpfunctions

word_equality_induction : tactic
word_integer_induction : tactic

```
;-------------------------------------------------------------------------------------+
       words_rules_compute.ml                   PRL_extensions for strings             |
--------------------------------------------------------------------------------------%

I.  RULES                        ( #  to be written/  T  uses a theorem/  +  needs improvement)

Id: Computation
----------------
  H >> ε∘v = w in WORDS by wreduce 1                                eps_concat_left no
     >> v = w in WORDS

  H >> v∘ε = w in WORDS by wreduce 1                                eps_concat_right no
     >> v = w in WORDS

  H >> ⊗a∘v = w in WORDS by wreduce 1                               sym_concat no
     >> a.v = w in WORDS

  H >> (a.u∘v) = w in WORDS by wreduce 1                            cons_concat no
     >> a.(u∘v) = w in WORDS

  H >> (u∘v)∘z = w in WORDS by wreduce 1                            con_assoz no
     >> u∘(v∘v) = w in WORDS
     >> u in WORDS
     >> v in WORDS
     >> w in WORDS


  H >> ←ε = w in WORDS by wreduce 1                                 eps_rev no
     >> ε = w in WORDS

  H >> ←⊗a = w in WORDS by wreduce 1                                sym_rev no
     >> ⊗a = w in WORDS

  H >> ←(a.u) = w in WORDS by wreduce 1                             cons_rev no
     >> ←u*a  = w in WORDS

  H >> ←(u∘v) = w in WORDS by wreduce 1                             rev_con no
     >> ←v∘←u = w in WORDS
     >> u in WORDS
     >> v in WORDS

  H >> ←(u*a) = w in WORDS by wreduce 1                             anticons_rev no
     >> a.←u = w in WORDS
     >> u in WORDS
     >> a in SYMBOLS

  H >> ←(←u) = w in WORDS by wreduce 1                              doublerev no
     >> u = w in WORDS


  H >> hd(a.l) = b in SYMBOLS by wreduce 1                          hd_up no
     >> a = b in SYMBOLS

  H >> tl(ε) = w in WORDS by wreduce 1                              tl_base no
     >> ε = w in WORDS

  H >> tl(a.v) = w in WORDS by wreduce 1                            tl_up no
     >> v = w in WORDS


  H >> |ε| = x in int by wreduce1                                   lg_base no
     >> 0 = x in int

  H >> |a.l| = x in int by wreduce1                                 lg_up no
     >> |l|+1 = x in int

  H >> |⊗a| = x in int by wreduce1                                  lg_1 no
     >> 1 = x in int
```

```
H >> |u∘v| = x in int by wreduce 1                          lg_concat no
    >> u in WORDS
    >> v in WORDS
    >> |u|+|v| = x in int

H >> |u*a| = x in int by wreduce 1                          lg_anticons no
    >> u in WORDS
    >> a in SYMBOLS
    >> |u|+1 = x in int

H >> |←v| = x in int by wreduce 1                           lg_rev no
    >> v in WORDS
    >>|v| = x in int

H >> |tl(v)| = x in int by wreduce 1                        lg_tl no
    >> v in WORDS
    >> v≠ε
    >> |v|-1 = x in int

H >> |v[l+1..lg]| = x in int by wreduce 1                   lg_pre no
    >> |v|-l = x in int
    >> v in WORDS

H >> |v[1..r]| = x in int by wreduce 1                      lg_suf no
    >> r = x in int
    >> v in WORDS

H >> |v[l+1..r]| = x in int by wreduce 1                    lg_range no
    >> r-l = x in int
    >> l < r+1


H >> v↑m = w in WORDS by                                    iter_down no    (NOT INCLUDED in wreduce !
    >> ε = w in WORDS
    >> m < 0

H >> v↑0 = w in WORDS by wreduce 1                          iter_base no
    >> ε = w in WORDS

H >> v↑i = w in WORDS by wreduce 1                          iter_up  no
    >> v∘(v↑i-1) = w in WORDS
    >> 0 < i

H >> v↑1 = w in WORDS by wreduce 1                          iter_1 no
    >> v = w in WORDS


H >> v[0+1..lg] = w in WORDS by wreduce 1                   pre_base no
    >> v = w in WORDS

H >> v[k+1..lg] = w in WORDS by wreduce 1                   pre_up no
    >> tl(v[(k-1)+1..lg]) = w in WORDS
    >> 0<k

H >> v[k+1..lg] = w in WORDS                                pre_down no     (NOT INCLUDED in wreduce !
    >> tl(v[(k+1)+1..lg]) = w in WORDS
    >> k<0

H >> tl(v[k+1..lg]) = w in WORDS by wreduce 1               pre_tl no
   >> (tl(v))[k+1..lg] = w in WORDS
   >> v in WORDS
   >> k in N

H >> v[|v|+1..lg] = w in WORDS by wreduce 1                 pre_top no
    >> ε = w in WORDS
    >> v in WORDS
```

```
H >> v(1) = b in SYMBOLS by wreduce 1                         sel_1 no
   >> hd(v) = b in SYMBOLS
   >> v in WORDS

H >> (a.v)(i+1) = b in SYMBOLS by wreduce 1                   sel_cons no
   >> v(i) = b in SYMBOLS
   >> a in SYMBOLS
   >> 0 < i+1

H >> v[1..m] = w in WORDS by                                  suf_down no    (NOT INCLUDED in wreduce !
   >> ε = w in WORDS
   >> m < 0

H >> v[1..0] = w in WORDS by wreduce 1                        suf_base no
   >> ε = w in WORDS

H >> v[1..r] = w in WORDS by wreduce 1                        suf_up no
   >> (v[1..r-1]∗v(r)) = w in WORDS
   >> 0 < r

H >> v[1..|v|] = w in WORDS by wreduce 1                      suf_top no
   >> v = w in WORDS

H >> v[0+1..r] = w in WORDS by wreduce 1                      range_suf no
   >> v[1..r] = w in WORDS

H >> v[1..r] = w in WORDS by                                  suf_range no   (NOT INCLUDED in wreduce !
   >> v[0+1..r] = w in WORDS

H >> v[l+1..|v|] = w in WORDS by wreduce 1                    range_pre no
   >> v[l+1..lg] = w in WORDS

H >> v[l+1..lg] = w in WORDS by                               pre_range no   (NOT INCLUDED in wreduce !
   >> v[l+1..|v|] = w in WORDS

H >> v(l).v[l+1..r] = w in WORDS by                           range_extend_left no
   >> v[l-1+1..r] = w in WORDS                                               (NOT INCLUDED in wreduce !

H >> v[l-1+1..r] = w in WORDS by wreduce 1                    range_split_left no
   >>v(l).v[l+1..r] = w in WORDS

H >> v[l+1..r]∘⊗v(r+1) = w in WORDS by                        range_extend_right no
   >> v[l+1..r+1] = w in WORDS                                               (NOT INCLUDED in wreduce !

H >> v[l+1..r+1] = w in WORDS by wreduce 1                    range_split_right no
   >> v[l+1..r]∘⊗v(r+1) = w in WORDS

H >> v[r+1..r] = w in WORDS by wreduce 1                      range_eps no
   >> ε = w in WORDS
   >> v in WORDS

H >> v[0+1..|v|] = w in WORDS by wreduce 1                    range_top no
   >> v = w in WORDS

H >> v[l+1..k]∘v[k+1..r] = w in WORDS by wreduce 1            range_concat no
   >> v[l+1..r] = w in WORDS
   >> l in int
   >> k in int
   >> r in int
   >> 0≤l
   >> l≤k
   >> k≤r
```

CONVERSIONS (to be written)

```
H >> ∃z:WORDS. ∃b:SYMBOLS. z*b = w in WORDS
    >> w≠ε

H >> w≠ε
    >> ∃z:WORDS. ∃b:SYMBOLS. z*b = w in WORDS

H >> |w| = 0 in int
    >> w = ε in WORDS

H >> w = ε in WORDS
    >> |w| = 0 in int

H >> 0 < |w|
    >> w≠ε

H >> w≠ε
    >> 0 < |w|

H >> w≠ε
    >> ¬ (w = ε in WORDS)

H >> ¬(w = ε in WORDS)
    >> w≠ε
```

Helpfunction

ordered_exp_list: int -> proof -> term list
if the conclusion is an equal term "a=b in T"
return [a;b] if the integer is 1 [b;a] otherwise

```
;----------------------------------------------------------------------------------------+
       words_tactics.ml                               PRL_extensions for strings          |
-----------------------------------------------------------------------------------------%

II: Tactics
------------

IIa: general tactics
--------------------
  Wintro  : tactic                  repeated word_introduction
  Wmember : tactic                  general membership tactic including knowledge about WORDS


IIb: All Introduction and Induction
-----------------------------------
  H >> ∀a:SYMBOLS.T                                               symbol_all_intro
     a:SYMBOLS >> T

  H >> ∀w:WORDS.T                                                 word_all_intro
     w:WORDS >> T

  H >> ∀w:WORDS.T                                                 word_induction
     >> T[ε/w]
     hd:SYMBOLS, tl:WORDS, tl_hyp:T[tl/w] >> T[hd.tl/w]

  H >> ∀w:WORDS.T                                                 word_tail_induction
     u:WORDS >> T[u/w] in U10
     >> T[ε/w]
     a:SYMBOLS, v:WORDS, T[v/w] >> T[v*a/w]

  H >> ∀w:WORDS.T                                                 word_lg_induction
     u:WORDS >> T[u/w] in U10
     >> T[ε/w]
      i:int, 0<i, (∀v:WORDS. |v|=i-1 in int => T[v/w]), u:WORDS, |u|=i in int
     >> T[u/w]

 If T[u/w] in U10 is provable by  tac  resp.  Wmember  use:

  H >> ∀w:WORDS.T                                                 tail_induction_using tac
     >> T[ε/w]                                                  / tail_induction
     a:SYMBOLS, v:WORDS, T[v/w] >> T[v*a/w]

  H >> ∀w:WORDS.T                                                 lg_induction_using tac
     >> T[ε/w]                                                  / lg_induction
      i:int, 0<i, (∀v:WORDS. |v|=i-1 in int => T[v/w]), u:WORDS, |u|=i in int
     >> T[u/w]


IIc: Some_introduction for WORDS
--------------------------------
  H >> ∃w1,..,wn:WORDS.T                                           word_some_intro level [v1;..;vn]
     w1,..,wn:WORDS >> T in Ui
     >> T[v1,..,vn/w1,..wn]                                  ("vi in WORDS" must be provable by Wmember)


IId: Noteps Introduction
------------------------
  H >> a.l≠ε                                                       word_intro_noteps    / wnoteps
     >> a in SYMBOLS
     >> l in WORDS

IIe: a special variant of con_assoz
-----------------------------------
  H >> (u∘v)∘w = x∘(y∘z) in WORDS     by conassoz 1
     >> u = x in WORDS
     >> v = y in WORDS
     >> w = z in WORDS
```

```
%-----------------------------------------------------------------------------------------+
|      words_recursion.ml    PRL_extensions for strings & recursive function definition    |
+-----------------------------------------------------------------------------------------%

I: Constructors, destructors & predicates
-----------------------------------------
   make_trk_term     = term -> term -> term       g h     into    "TRK(g,h)"
   make_idtrk_term   = term -> term               h       into    "h*"

   make_trktype_term = term -> term -> term       A B     into    "(A#WORDS) -> B"

 destructors & predicates as usual


I: RULES
--------

Ib: Introduction
----------------
  H >> TRK(g,h) in (A#WORDS) -> B                              trk_intro
     >> A in TYPE
     >> B in TYPE
     >> g in A->B
     >> h in (B#SYMBOLS) -> B

  H >> h* in (B#WORDS) -> B                                     trkid_intro
     >> B in TYPE
     >> h in (B#SYMBOLS) -> B

  H >> h*(x,w) in B                                             trkid_equality_apply
     >> B in TYPE
     >> h in (B#SYMBOLS) -> B
     >> x in B
     >> w in WORDS


Id: Computation                                                 ( # not written yet)
---------------
  H >> TRK(g,h)(x,ε) = t in B  by trk_reduce 1                  trk_reduce_base no
     >> g(x) = t in B

  H >> TRK(g,h)(x,v∘⊗a) = t in B by trk_reduce 1                trk_reduce_up no
     >> h(TRK(g,h)(x,v),a) = t in B

  H >> h*(x,ε) = t in B  by trk_reduce 1                        trkid_reduce_base no
     >> x = t in B

  H >> h*(x,v*a) = t in B by trk_reduce 1                       trkid_reduce_up no
     >> h(h*(x,v),a) = t in B
     (>> v in WORDS
      >> a in SYMBOLS
      >> B in TYPE
      >> x in B
      >> h in (B#SYMBOLS) -> B)

  H >> h*(x,u∘v) = t in B by trk_reduce 1                       trkid_reduce_concat no
     >> h*(H*(x,u),v) = t in B
     >> v in WORDS
     >> w in WORDS
     >> B in TYPE
     >> x in B

II Tactics
----------
     Try to prove the unnecessary subgoals of star_reduction, dfa_equality_apply

     trkid_anticons_compute : int -> tactic
     trkid_concat_compute   : int -> tactic
     trkid_apply            : tactic
```

```
--------------------------------------------------------------------------------------+
      dfa.ml                  PRL_extensions for      FINITE AUTOMATA                 |
--------------------------------------------------------------------------------------%
```

. Constructors, destructors & predicates

a: Constructors

```
  make_states_term/states = term                                                 "STATES"
  make_dfa_term   /dfa    = term                                                 "DFA"
  make_aut_term           = term -> term -> term -> term -> term   Q d q0 F  into  "(Q,d,q0,F)"
  make_star_term          = term -> term                           d         into  "d*"
  make_accept_term        = term -> term                           M         into  "L(M)"
  make_accepted_term      = term -> term -> term                   w M       into  "w εL(M)"
```

Additional functions

```
  make_table_type_term    = term -> term                           Q         into  "({Q}#SYMBOLS) -> {Q}"
```

Destructors & Predicates as usual

I. RULES   (# unfinished)

Helpfunction:

```
   H, Q:STATES, H' >> (({Q}#SYMBOLS)->{Q}) # {Q} # P({Q}) in Ui   by dfa_parts_intro   (i>1)
```

Ia: Formation

```
  H >> STATES in Ui by dfintro                                      state_intro

  H >> {Q} in Ui by dfintro                                         state_set_intro
     >> Q in STATES

  H >> DFA in Ui by dfintro     (i>1)                               dfa_intro

  H >> (w ε L(M)) in Ui by dfintro                                  accepted_intro
     >> M in DFA
     >> w in WORDS

  H >> L(M) in Ui by dfintro                                        accept_intro
     >> M in DFA
```

Ib: Introduction

```
  H >> (Q,d,q0,qf) in DFA by dfintro                                dfa_equality
     >> Q in STATES
     >> d in ({Q}#SYMBOLS) -> {Q}
     >> q0 in {Q}
     >> qf in {Q}

  H >> d* in ({Q}#WORDS) -> {Q} by dfintro                          star_intro
     >> Q in STATES
     >> d in ({Q}#SYMBOLS) -> {Q}

  H >> w in L(M) by dfintro                                         accept_equality
     >> w in WORDS
     >> d*(q0,w) = qf in {Q}  -where (Q,d,q0,qf) = M-

  H >> d*(q,w) in T by dfintro                                      delstar_equality_apply
     >> T in TYPE
     >> d in (T#SYMBOLS) -> T
     >> q in T
     >> w in WORDS
```

```
Ic: Elimination
----------------
  H, M:DFA, H` >> T by dfelim M                                        dfa_elim hyp
     Q:STATES, d:({Q}#SYMBOLS) -> {Q}, q0:{Q}, F: P({Q}), M=(Q,d,q0,F) in DFA
     >> T[(Q,d,q0,F)/M]

  H >> t -where (Q,d,q0,qf) = M- in T[M/z] by dfintro                  dfa_equality_spread
     >>M in DFA
      Q':STATES, d':({Q'}#SYMBOLS) -> {Q'}, q0':{Q'}, qf':{Q'}, M=(Q',d',q0',qf') in DFA
     >> t[Q',d',q0',qf'/Q,d,q0,qf] in T[(Q',d',q0',qf')/z]

  H, w:L(M), H' >> T by dfelim w                                       accept_elim hyp
     >> M in DFA
     v:WORDS, v ε L(M), v = w in WORDS >> T[v/w]

Id: Computation
----------------
  H >> t -where (Q,d,q0,qf) = (Q',d',q0',qf')- = t' in T by dfreduce1  dfa_reduction no
     >> t[Q',d',q0',qf'/Q,d,q0,qf] = t' in T

  H >> d*(q,ε) = q' in {Q} by dfreduce 1                               star_reduction_eps no
     >> q = q' in {Q}

  H >> d*(q,v∘⊗a) = q` in {Q} by dfreduce 1                            star_reduction_anticons no
     >> d(d*(q,v),a) = q` in {Q}
     (>> v in WORDS
      >> a in SYMBOLS
      >> Q in STATES
      >> q in {Q}
      >> d in ({Q}#SYMBOLS) -> {Q})

  H >> d*(q,v∘w) = q' in {Q} by dfreduce 1 CONCAT                      star_reduction_concat no
     >> d*(d*(q,v),w) = q' in {Q}
     (>> v in WORDS
      >> w in WORDS
      >> Q in STATES
      >> q in {Q}
      >> d in ({Q}#SYMBOLS) -> {Q})

II Tactics
-----------
  H >> t -where (Q,d,q0,qf) = (Q',d',q0',qf')-  by                      dfa_computation
     >> t[Q',d',q0',qf'/Q,d,q0,qf]

  H, t -where (Q,d,q0,qf) = (Q',d',q0',qf')-, H' >> T by                dfa_computehyp no
     H, t[Q',d',q0',qf'/Q,d,q0,qf], H' >> T

All-introduction
   H >> ∀Q:STATES. T                                                    states_all_intro
     Q:STATES >> T

   H >> ∀M:DFA. T                                                       dfa_all_intro
     M:DFA >> T

   H >> ∀w:L(M). T                                                      accept_all_intro
     w:L(M) >> T

   H >> ∀M:DFA. T                                                       dfa_intro_elim
      M:DFA, Q:STATES, d:({Q}#SYMBOLS)->{Q}, q0:{Q}, F:P({Q}), M=(Q,d,q0,F) in DFA
     >> T[(Q,d,q0,F)/M]

   H >> ∀w:L(M). T                                                      accept_intro_elim
     w:L(M) >> M in DFA
     w:L(M), v:WORDS, v ε L(M), v = w in WORDS >> T[v/w]

Try to prove the unnecessary subgoals of star reduction, dfa equality apply
    star_anticons_compute no
    star_concat_compute no
    delstar_apply
```