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. 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\times B (cartesian products), A\uplus B (disjoint union) and A \rightarrow B (the functions from A to B). In addition to these constructors there are dependent products x:A\times B (resembling \Sigma x:A. B(x)), which expresses that the type of the second member b of a pair \langle a, b \rangle can depend on the value of a, and dependent functions x:A \rightarrow B ((\Pi x:A. B(x))). Set types \{x:A\mid 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 propositions can formally be expressed as types: 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\neq 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 x1:T1, . . . , xn:Tn \rightarrow S where xi are variables and Ti and S are terms. x1:T1, . . . , xn:Tn 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

\begin{align*}
H \rightarrow T & \text{ by \langle rule-name\rangle} \\
H_1 & \rightarrow T_1 \\
& \quad \quad \vdots
H_k & \rightarrow T_k
\end{align*}

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 \((\lambda x. b(x))(a)\) into their canonical form \(b(a)\)) and some miscellaneous rules not associated with a particular type (e.g. \texttt{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 (\texttt{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 extremely 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 occurring 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:

\[
Q \text{: STATES } \
\text{S} \text{: ALPHABETS } \# (Q \# S \rightarrow Q) \# Q \# P(Q)
\]

For convenience I took the alphabet \(S\) out of this definition and replaced it by a general alphabet \textsc{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

- \textbf{finite sets} including a version of the pigeonhole principle which is needed for the proof of the pumping-lemma

- \textbf{sets} in general, mainly the powerset construct

- \textbf{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 \textsc{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 PRL 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 planning 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 \( B[a_1, \ldots, a_n/x_1, \ldots, x_n] \) where one of the hypotheses or a theorem was \( \forall x_1: A_1, \ldots, \forall x_n: A_n. B \) 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 dependecies 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 member (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

\[
\begin{align*}
(*\text{comment}*) & = \langle t \rangle \text{ THEN autotactic} \\
\& \langle t \rangle & = \langle t \rangle \text{ THENMEMBER } \langle t \rangle \text{ member_tac} \\
@ \langle t \rangle & = \langle t \rangle \text{ THEN member_tac} \\
\text{THEOREM } \langle \text{name} \rangle & = \text{thm 'name'}
\end{align*}
\]
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*), *auto tactic* 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 these 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 \( \forall \)
and \( \exists \) instead of the names *all* and *some*.

\[
\langle p; \text{prop} \rangle \land \langle q; \text{prop} \rangle = (\langle p \rangle \# \langle q \rangle)
\]

\[
\forall x: \text{var} \rangle ; \langle t; \text{type} \rangle . \langle p; \text{prop} \rangle = (\langle x \rangle : \langle t \rangle \rightarrow \langle \langle p \rangle \rangle)
\]

\[
\forall x: \text{var}, y: \text{var} \rangle ; \langle t; \text{type} \rangle . \langle p; \text{prop} \rangle = \forall \langle x \rangle : \langle t \rangle . \forall \langle y \rangle : \langle t \rangle . \langle p \rangle
\]

\[
\forall x: \text{var}, y: \text{var}, z: \text{var} \rangle ; \langle t; \text{type} \rangle . \langle p; \text{prop} \rangle = \forall \langle x \rangle : \langle t \rangle . \forall \langle y \rangle : \langle t \rangle . \forall \langle z \rangle : \langle t \rangle . \langle p \rangle
\]

\[
\exists x: \text{var} \rangle ; \langle t; \text{type} \rangle . \langle p; \text{prop} \rangle = (\langle x \rangle : \langle t \rangle \# \langle p \rangle)
\]

\[
\exists x: \text{var}, y: \text{var} \rangle ; \langle t; \text{type} \rangle . \langle p; \text{prop} \rangle = \exists \langle x \rangle : \langle t \rangle . \exists \langle y \rangle : \langle t \rangle . \langle p \rangle
\]

\[
\exists x: \text{var}, y: \text{var}, z: \text{var} \rangle ; \langle t; \text{type} \rangle . \langle p; \text{prop} \rangle = \exists \langle x \rangle : \langle t \rangle . \exists \langle y \rangle : \langle t \rangle . \exists \langle z \rangle : \langle t \rangle . \langle p \rangle
\]

\[
\langle p; \text{prop} \rangle \rightarrow \langle q; \text{prop} \rangle = (\langle p \rangle \rightarrow \langle q \rangle)
\]

\[
\langle p; \text{prop} \rangle \leftrightarrow \langle q; \text{prop} \rangle = \langle p \rangle \Rightarrow \langle q \rangle \land \langle q \rangle \Rightarrow \langle p \rangle
\]

\[
\text{false} = \text{void}
\]

\[
\neg \langle p; \text{prop} \rangle = (\langle p \rangle \neg \text{false})
\]

\[
\text{TYPE} = \text{unit}
\]

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 \( H \gg 3x: A_1 \ldots 3x: A_n . B \) by giving witnesses \( a_1, \ldots, a_n \) step by step would result in 
the following subgoals:

\[
\gg a_1 \text{ in } A_1
\]

\[
\gg a_n \text{ in } A_1[x_1, \ldots, x_n] \text{ in } A_n[a_1, \ldots, a_{n-1} / x_1, \ldots, x_{n-1}]
\]

\[
B[a_1, \ldots, a_n / x_1, \ldots, x_n] \text{ in } U_1
\]

\[
x_1 : A_1 \gg (\exists x_2 : A_2, \ldots, \exists x_n : A_n . B) \text{ in } U_1
\]

In order to prove the last \( n \) subgoals one would now have to perform a lot of introduction steps and 
then prove \( B[a_1, \ldots, a_j / x_1, \ldots, x_j] \text{ in } U_1 \) for \( 1 \leq j \leq n-1 \), \( A_n[a_1, \ldots, a_j / x_1, \ldots, x_j] \text{ in } U_1 \) for \( 1 \leq j \leq n-2 \) etc.

By sequencing goals like \( \forall x_1 : A_1 \ldots \forall x_n : A_n . (B \text{ in } U_1) \) 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:

\[
H \gg \exists x_1:A_1..\exists x_n:A_n.B \text{ by } repeat\_some\_intro \ i \ [a_1:..;a_n]
\]

\[
\gg A_1 \text{ in } U_1
\]

\[
x_1:A_1 \gg A_2 \text{ in } U_1
\]

\[
x_1:A_1..x_n:A_n \gg B \text{ in } U_1
\]

\[
\gg a_1 \text{ in } A_1
\]

\[
\gg x_n \text{ in } A_n[a_1,..,a_{n-1}/x_1,..,x_{n-1}]
\]

\[
\gg B[a_1,..,a_n/x_1,..,x_n]
\]

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 separately. The tactic `AllIntro`, for example, reduces the number of subgoals after introducing a universal quantifier to one

\[
H \gg \forall x:A.B \text{ by } AllIntro \ j \ AUjtc\ x:A \gg B
\]

provided that `AUjtc` is able to prove `A` in `U` 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-specific subgoals exist for introduction of existence problems, implication and equivalence.

### 3.3 Tupling

(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.

\[
\langle a:\text{term}, b:\text{term} \rangle = \langle \langle a, b \rangle \rangle
\]

\[
\langle a:\text{term}, b:\text{term}, c:\text{term} \rangle = \langle \langle a, (\langle b, c \rangle) \rangle \rangle
\]

\[
\langle a:\text{term}, b:\text{term}, c:\text{term}, d:\text{term} \rangle = \langle \langle a, (\langle b, c, d \rangle) \rangle \rangle
\]

\[
\langle t:\text{term} \rangle -\text{where } \langle a: \text{id}, b: \text{id} \rangle = \langle x:\text{term} \rangle -\text{where } \langle a: \text{id}, b: \text{id}, c: \text{id} \rangle = \langle x:\text{term} \rangle -\text{where } \langle a: \text{id}, b: \text{id}, c: \text{id}, d: \text{id} \rangle = \langle x:\text{term} \rangle -\text{where } \langle a: \text{id}, b: \text{id}, c: \text{id}, d: \text{id} \rangle = \langle x:\text{term} \rangle
\]

\[
\langle t: \text{term} \rangle -\text{where } \langle b: \text{term}, c: \text{term}, d: \text{term} \rangle = \langle t: \text{term} \rangle -\text{where } \langle b: \text{term}, c: \text{term}, d: \text{term} \rangle = \langle t: \text{term} \rangle -\text{where } \langle b: \text{term}, c: \text{term}, d: \text{term} \rangle = \langle t: \text{term} \rangle
\]

\[
\langle x: \text{id}, y: \text{id}, t: \text{term} \rangle -\text{where } \langle x: \text{id}, y: \text{id} \rangle = \langle x: \text{id}, y: \text{id} \rangle
\]

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 complethyp.
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 \( (\text{sets.m1}) \)

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 \( A \cap B = \{x\mid x \in A \land x \in B\} \). However in type theory (and also in order to avoid Russell’s paradox) 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 \( x \in A \) 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 \( \langle \text{carrier}, \text{pred} \rangle \) where \( \text{carrier} \) gives the context and \( \text{pred} \) is a predicate on \( x \in A \). In analogy to conventional set-notation the definition \( \langle x; \text{carrier} \mid \text{pred} \rangle \) stands for the pair \( \langle \text{carrier}, \lambda x. \text{pred} \rangle \). A conversion from pairs into a set-type of PRL is necessary if access to the elements of a \( \text{SET} \) 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**

\[
\begin{align*}
\langle x;\text{id}\rangle; \langle \text{carrier}; \text{TYPE} \rangle &; \langle \text{pred}; \text{TYPE} \rangle == \langle \text{carrier}, \langle \lambda x. \text{pred} \rangle \rangle \\
\langle \text{el}; \text{element} \rangle; \langle \text{carrier}; \text{TYPE} \rangle == \langle \text{el}; \langle \text{carrier} \rangle | \text{el} \in \langle \text{carrier} \rangle \rangle \\
\langle x; \text{SETS} \rangle &; \text{pred}; \langle \text{X}; \text{SETS} \rangle == \text{carrier} -\text{where} \ (\text{carrier}, \text{pred}) \langle x \rangle - \\
\langle \text{el}; \text{element} \rangle \in \langle x; \text{SETS} \rangle &; \text{pred}; \langle \text{X}; \text{SETS} \rangle == \text{pred}(\langle x \rangle)(\langle x \rangle) \\
\langle x; \text{SETS} \rangle &; \text{into a set} \rangle == \langle x; \text{carrier}; \text{pred}(x) \rangle -\text{where} \ (\text{carrier}, \text{pred}) \langle x \rangle - \\
\langle \text{T}; \text{TYPE} \rangle == \langle \text{S}; \text{SETS} \rangle \text{carrier}(\text{S}) -\langle \text{T} \rangle \text{ in} \ \text{TYPE} \\
\end{align*}
\]

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

\[
\begin{align*}
\text{A} \cup \text{B} &; == \langle x; \text{carrier}(A) | (x \in A | x \in B) \rangle \quad \text{(Union)} \\
\text{A} \cap \text{B} &; == \langle x; \text{carrier}(A) | (x \in A \land x \in B) \rangle \quad \text{(Intersection)} \\
\text{A} - \text{C} &; == \langle x; \text{carrier}(A) | -x \in A \rangle \quad \text{(Complement)} \\
\text{A} \subseteq \text{B} &; == \forall x; \text{carrier}(A); x \in A = x \in B \quad \text{(Subset)} \\
\text{P}(A) &; == \{S; \text{SETS} | \text{S(A)} \} \quad \text{(alternate powerset definition)}
\end{align*}
\]

where \( A \) and \( B \) are \( \text{SETS} \) 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 \( \text{SETS} \) 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 problems similar to the ones of general set theory. Given a finite set $\mathcal{Q}$ one is interested in its cardinality - a proof for being finite - and in access to its elements. Since the Nuprl set-constructor \( \langle x: A \mid B \rangle \) generally does not allow to use the information $B$ in proofs, a definition of \textsc{finite sets} as a subset of \textsc{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 $\mathcal{Q}$ 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 $\{1, \ldots, n\}$.

The rules corresponding to these definitions

\[
\langle i: \text{int} \rangle \leq \langle j: \text{int} \rangle \quad \implies \quad (\langle i \rangle - 1) < (\langle j \rangle)
\]

\[
\langle x_1: \text{int} \rangle \neq \langle x_2: \text{int} \rangle \quad \implies \quad \neg (\langle x_1 \rangle = (\langle x_2 \rangle)) \in \text{int}
\]

\[
\mathcal{N} \quad \{1, \ldots, \langle \text{bound}: \text{N} \rangle \} \quad \{0, \ldots, \langle r: \text{N} \rangle \} \quad \{\text{num: int} \mid -1 < \text{num} \}
\]

\[
\text{FINITE SETS} \quad \langle \mathcal{A} : \text{FINITE SETS into a set} \rangle \quad \{1, \ldots, \langle \mathcal{A} \rangle \}
\]

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:

* \textit{fineq}
  
  \[
  \text{THM} \implies \forall m: \text{N}. \ \forall x: \{1, \ldots, (m-1)+1\}. \ x \in \{1, \ldots, m\}
  \]

* \textit{extend}
  
  \[
  \text{THM} \implies \forall k: \text{N}. \ \forall y: \{1, \ldots, k\}. \ y \in \{1, \ldots, k+1\}
  \]

* \textit{restrict}
  
  \[
  \text{THM} \implies \forall m: \text{N}. \ \forall x: \{1, \ldots, m\}. \ x < m \implies x \in \{1, \ldots, m-1\}
  \]

* \textit{restrict1}
  
  \[
  \text{THM} \implies \forall m: \text{N}. \ \forall x: \{1, \ldots, m\}. \ x = m \implies x \in \{1, \ldots, m-1\}
  \]

* \textit{finarith}
  
  \[
  \text{THM} \implies \forall m: \text{N}. \ \forall x: \{1, \ldots, m\}. \ (x = m \in \{1, \ldots, m\}) \mid (x < m)
  \]

* \textit{hole1}
  
  \[
  \text{THM} \implies \forall k: \text{int}. \ 0 \leq k \implies \forall n: \text{N}. \ \forall f: \{1, \ldots, k\} \rightarrow \{1, \ldots, n\}. \ \forall y: \{1, \ldots, n\}
  
  \exists x: \{1, \ldots, k\}. \ f(x) = y \in \text{int} \mid \forall x: \{1, \ldots, k\}. \ f(x) \neq y
  \]

- 10 -
Some of them deal with problems of integer arithmetic the \texttt{arith}\_rule should be able to handle.

\begin{itemize}
    \item \texttt{Int\_arith}
        \begin{verbatim}
        THM>\forall x,y: int. (x = y in int) | x\&y
        \end{verbatim}
    \item \texttt{Int\_sub}
        \begin{verbatim}
        THM>\forall i,j: int. i\&j in int \rightarrow j\&i = 0 in int
        \end{verbatim}
    \item \texttt{Less\_arith}
        \begin{verbatim}
        THM>\forall i,j: int. i\&j \rightarrow 0\&j\&i
        \end{verbatim}
    \item \texttt{Less\_arith1}
        \begin{verbatim}
        THM>\forall i,j,k: int. i\&j \& 0\&k \rightarrow i\&j\&k
        \end{verbatim}
\end{itemize}

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.

\begin{itemize}
    \item \texttt{Pigeon}
        \begin{verbatim}
        THM>\forall k: int. 0\&k \rightarrow \forall f:(1\&\ldots,k+1) \rightarrow \{1\&\ldots,k\}. \exists i,j:(1\&\ldots,k+1). i\&j \& f(i)=f(j) in int
        \end{verbatim}
\end{itemize}

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 \texttt{membership} shows that the down case \((k=0)\) contradicts the assumption \(0\&k\).

```
EDIT THM pigeon
* top
    \begin{verbatim}
    \forall k: int. 0\&k \rightarrow \forall f:[1\&\ldots,k+1] \rightarrow \{1\&\ldots,k\}. \exists i,j:[1\&\ldots,k+1]. i\&j \& f(i)=f(j) in int
    \end{verbatim}
    BY
    \begin{verbatim}
    int_all_intro
    THEN @Elim 1
    THENL [ -Seq [FALSE]; Elim 3 THENL [membership; Thinning [3,4]] ]
    \end{verbatim}
    1* 1. k: int
    2. 0\&k
    3. f:[1\&\ldots,k+1] \rightarrow \{1\&\ldots,k\}
    \begin{verbatim}
    >> false
    \end{verbatim}
    2* 1. k: int
    2. 0\&k
    3. f:[1\&\ldots,k+1] \rightarrow \{1\&\ldots,k\}
    \begin{verbatim}
    \forall f:[1\&\ldots,(k-1)+1] \rightarrow \{1\&\ldots,(k-1)\}. \exists i,j:[1\&\ldots,(k-1)+1]. i\&j \& f(i)=f(j) in int
    \end{verbatim}
    \begin{verbatim}
    >> \exists i,j:[1\&\ldots,k+1]. i\&j \& f(i)=f(j) in int
    \end{verbatim}
```

The base case \((k=0)\) is easy because there is no function \(f:[1\&\ldots,0+1] \rightarrow \{1\&\ldots,0\}\).

```
EDIT THM pigeon
* top 1
    \begin{verbatim}
    1. k: int
    2. 0\&0
    3. f:[1\&\ldots,0+1] \rightarrow \{1\&\ldots,0\}
    \end{verbatim}
    \begin{verbatim}
    >> false
    \end{verbatim}
    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 \( \{1, \ldots, k\} \) or \( f(x) = f(k+1) \) for all \( x \) in \( \{1, \ldots, k\} \)

```
EDIT THM pigeon
* top 2
1. k: int
2. \( \forall k \)
3. \( f: \{1, \ldots, k+1\} \to \{1, \ldots, k\} \)
4. \( \forall i: \{1, \ldots, (k+1)\} \to \{1, \ldots, k\} \). \( \exists j: \{1, \ldots, (k+1)\}. \, i < j & f(i) = f(j) \) in int

BY Cases [\( \exists x: \{1, \ldots, k\}. \, f(x) = f(k+1) \) in int'; \( \forall x: \{1, \ldots, k\}. \, f(x) = f(k+1) \)]
```

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. \( \forall k \)
3. \( f: \{1, \ldots, k+1\} \to \{1, \ldots, k\} \)
4. \( \forall i: \{1, \ldots, (k+1)\} \to \{1, \ldots, k\} \). \( \exists j: \{1, \ldots, (k+1)\}. \, i < j & f(i) = f(j) \) in int

BY theorem 'hole1' ['k':imp;'k';'\( \forall x, f(x) \);'f(k+1)']
```

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 \( \forall k: \text{nat}. \, \forall f: \{1, \ldots, k+1\} \to \{1, \ldots, k\}. \, \forall i,j: \{1, \ldots, k+1\}. \, (i < j & f(i) = f(j) \text{ in int}) \text{ in U1} \)
```

```
EDIT THM pigeon
* top 2 2
1. k: int
2. \( \forall k \)
3. \( f: \{1, \ldots, k+1\} \to \{1, \ldots, k\} \)
4. \( \forall i: \{1, \ldots, (k+1)\} \to \{1, \ldots, k\} \). \( \exists j: \{1, \ldots, (k+1)\}. \, i < j & f(i) = f(j) \) in int
5. \( \exists x: \{1, \ldots, k\}. \, f(x) = f(k+1) \) 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 \( g : (1, \ldots, k) \rightarrow (1, \ldots, k-1) \).

\( f \) itself will possibly map some \( x \) in \( (1, \ldots, k) \) to \( k \). Then however, because of the assumption in hypothesis 5, \( f(k+1) \) will be in \( (1, \ldots, k-1) \). Thus \( g(x) = (f(k+1) \text{ if } f(x) = k, f(k+1) \text{ otherwise}) \) will do.

The proof of this informal description requires some juggling with finite-set-arithmetic as performed in top 231. In preparation for the next step (top 232) also the induction hypothesis 4 is unraveled into the hypotheses 5-8.

```
EDIT THM pigeon
--------------------------
* top 2 3 1
1. k:int
2. 0<k
3. f:(1,...,k+1) \rightarrow (1,...,k)
4. \forall x : (1,...,(k-1)+1) \rightarrow [1,...,k-1]. \exists i,j:[1,...,(k-1)+1] x i<j \& f(i) = f(j) \in int
5. \forall x : (1,...,k). f(x) \neq f(k+1)

BY all_elim_on ['\x. int_eq(f(x);k;f(k+1);f(x))'] 4
THEN
[ 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. j:[1,...,(k-1)+1]
   >> int_eq(f(i);k;f(k+1);f(i)) = int_eq(f(j);k;f(k+1);f(j)) in int

EDIT THM pigeon
--------------------------
* top 2 3 1
1. k:int
2. 0<k
3. f:(1,...,k+1) \rightarrow (1,...,k)
4. \forall x : (1,...,k). f(x) \neq f(k+1)
5. x:[1,...,(k-1)+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 'f1' ['k';'f(x)']
   : Equal
   : THEOREM 'restrict1'
   THENL
 [ membership;membership;theorem 'finfuntype' ['k';'f';'k+1']
   : THEOREM 'restrict1' ] ] ]
```

-13-
Now again a case analysis has to be done:

\[ f(i) = k \text{ or } f(i) = k \text{ or } f(j) = k \text{ or } f(i) = k \text{ and } f(j) = k \]

The validity of these cases again is mainly a problem of finite-set-arithmetic. Trivial facts like \( f(i) \text{ in int} \) have to be sequenced first which makes the rule look more complicated than it actually is.
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 3
1. k: int
2. 0 < k
3. f: [1, ..., k] → [1, ..., k]
4. ∀x: [1, ..., k]. f(x) ≠ f(k + 1)
5. ∀x: [1, ..., (k - 1) + 1]. f(x) ≠ f(k + 1)
6. j: [1, ..., (k - 1) + 1]
7. < 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
11. f(i) ≠ k
12. f(j) ≠ k
13. f(j) ≠ k

BY (* choose i = j 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 ]
    ; 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'] 0 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. f: [1, ..., (k - 1) + 1]
6. j: [1, ..., (k - 1) + 1]
7. < 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

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
  THENL
  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(i) in int' ]
  THENL [-Inteq_computation 1 false; -Inteq_computation 1 false; Equal] ]

- 16 -
Proof-tree of the theorem "pigeon"

\[ \forall k \in \text{int} \ 0 \leq k \Rightarrow \forall f : \{1, \ldots, k+1\} \rightarrow \{1, \ldots, k\} \]
\[ \exists i, j : \{1, \ldots, k+1\}. \ i < j \land f(i) = f(j) \text{ in int} \]

1. **Induction on** \( k \).
   - The down case is contradictory.
   - The base case leads to false.

2. **Base Case:**
   - There is no function from \( \{1, \ldots, 1\} \) to \( \{1, \ldots, 0\} \).
   - Consider two cases:
     1. \( \exists x \in \{1, \ldots, k\}. f(x) = f(k + 1) \)
     2. \( \forall x \in \{1, \ldots, k\}. f(x) \neq f(k + 1) \)

3. **Apply Lemma 1** to show that either one of these two cases is true.
   - Choose \( i = x \) and \( j = k + 1 \).
   - 1. Use induction hypothesis to show
      \[ \text{int}_e q(f(i), k; f(k + 1), f(i)) = \text{int}_e q(f(j), k; f(k + 1), f(j)) \]
   - Consider four cases:
     1. \( f(i) = k \land f(j) = k \)
     2. \( f(i) = k \land f(j) \neq k \)
     3. \( f(j) = k \land f(j) = k \)
     4. \( f(i) \neq k \land f(j) = k \)

4. **Show that the function**
   \[ \forall x. \text{int}_e q(f(x), k; f(k + 1), f(x)) \]
   is of the required type.
   - Consider four cases:
     1. \( x = k \land f(j) = k \)
     2. \( x = k \land f(j) \neq k \)
     3. \( x \neq k \land f(j) = k \)
     4. \( x \neq k \land f(j) \neq k \)

   - **Choose** \( i \) and \( j \) as given.
   - **Choose** \( i = j \) and \( j = k + 1 \).
   - **Choose** \( i = i \) and \( j = k + 1 \).
   - **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 \texttt{WORDS} - which is just another name for strings - and all kind of operations on them. I chose the type \texttt{int} to represent the standard alphabet \texttt{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 \texttt{SYMBOLS} (except that \texttt{SYMBOLS} has to be in \texttt{ui}) they work on any other type as well. The fact that \texttt{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. \texttt{e} is the empty word; \texttt{aa} is the word solely consisting of the symbol \texttt{a}; \texttt{uw} is the concatenation of \texttt{u} and \texttt{v}; \texttt{aa} means inserting the symbol \texttt{a} at the end of the word \texttt{w} (cons at the tail of \texttt{w}); \texttt{w} is the word \texttt{w} in reverse order; \texttt{w} is the i-fold iteration of \texttt{w}, \texttt{hd(w)} is the first symbol of \texttt{w}, \texttt{tl(w)} is the rest, \texttt{|w|} is the length of \texttt{w}, \texttt{w[1..1]} is the substring of \texttt{w} resulting from cutting off the first \texttt{i} symbols; \texttt{w[i]} is the i-th symbol in \texttt{w}, \texttt{w[i..]} is the substring of \texttt{w} consisting of the first \texttt{i} symbols; \texttt{w[i..r]} is the word \texttt{w(i+1)...w(r)}.

\texttt{SYMBOLS} == \texttt{int}
\texttt{WORDS} == \texttt{SYMBOLS list}
\texttt{e} == \texttt{nil}
\texttt{<word:WORDS>=e} == \texttt{3a:SYMBOLS. \exists l:WORDS. <word> = a.l in WORDS}
\texttt{@<sym:SYMBOLS>} == (\texttt{:<sym>,e})
\texttt{<lw:WORDS><<rw:WORDS>} == \texttt{list_ind(<lw>;<rw>;hd1.tll.tll_rw.(hd1.tll_rw))}
\texttt{<word:WORDS>*<sym:SYMBOLS>} == (\texttt{:<word>*<sym>})
\texttt{<word:WORDS>} == \texttt{list_ind(<word>:c:hdword,t1word,t1revword:hdword)}
\texttt{<word:WORDS>!<ii:N>} == \texttt{ind(<ii>);k1,indhyp.etc:k1,indhyp.(<word>*indhyp)}
\texttt{hd(<word:WORDS>)} == \texttt{list_ind(<word>:0:hdword,t1word,hdword,hdword,hdword)}
\texttt{tl(<word:WORDS>)} == \texttt{list_ind(<word>:c:hdword,t1word,hdword,hdword,hdword)}
\texttt{|<word:WORDS>|} == \texttt{list_ind(<word>:0:hdword,t1word,lgtlword.(lgtlword+1))}
\texttt{<word:WORDS>!<[ii:1..1]>} == \texttt{ind(<ii>);k1.w_i_1g.w_i_1g;<word;k1,w_i_1g.tl(w_i_1g)}
\texttt{<word:WORDS>!<[ii:N]>} == \texttt{hd(<word>;[<ii>-<ii+1>..1]})
\texttt{<word:WORDS>!<[ii:1..1]>} == \texttt{ind(<ii>);k1.w_1_k1_1_1;e;k1.w_1_k1_1_1.w_1_k1_1*<word;k1>}
\texttt{<word:WORDS>!<[lb:N]+1,<rb:N]}} == \texttt{<word;([1..<rb>]<lb>+1..1]l}]
\texttt{TRK((g:A\to B),(h:B\#SYMBOLS\to \emptyset B))}
\texttt{<h1:B\#SYMBOLS\to \emptyset B>}} == \texttt{TRK((\lambda 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 \texttt{tail-recursion}. For \texttt{g:A\to B}, \texttt{h:B\#SYMBOLS\to \emptyset B} \texttt{TRK(g,h)} is the function \texttt{f:A\#WORDS\to B} defined by \texttt{f(x,e) = g(x) and f(x,\#a) = h(f(x,w),a)} for \texttt{e:A}, \texttt{a:B\#SYMBOLS} and \texttt{\#w:WORDS}. \texttt{h} is the same for \texttt{g} being the identity function. A Nuprl-proof of these properties is performed in the theorems \texttt{trktype - trk4} in the library automata.

*\texttt{trktype}*

\texttt{THM \gg A,B;TYPE. \forall g:A\to B. \forall h:(B\#SYMBOLS)\to B. TRK(g,h) in (A\#WORDS)\to B}

* This is one of the advantages of defining objects by \texttt{not using extract terms of theorems}.
• trk1
  THM \rightarrow \forall A, B. \forall h : (B \rightarrow B). \forall g : A \rightarrow B. \forall x : A. \forall y : B. \mathbf{TRK}(g, h)(x, z) = g(x) in B

• trk2
  THM \rightarrow \forall A, B. \forall h : (B \rightarrow B). \forall A. \forall x : A. \forall w : \text{WORDS}. \forall z : \text{SYMBOLS}
  . \mathbf{TRK}(g, h)(x, w, a) = h(\mathbf{TRK}(g, h)(x, w), a) in B

• trk3
  THM \rightarrow \forall A, B. \forall h : (B \rightarrow B). \forall A. \forall w : \text{WORDS}. \forall z : \text{SYMBOLS}
  . \mathbf{TRK}(g, h)(x, w) = h^*(g(x), w) in B

• trk4
  THM \rightarrow \forall A, B. \forall h : (B \rightarrow B). \forall u, v : \text{WORDS}. \forall z : \text{SYMBOLS}
  . h^*(x, (u+v)) = h^*(h^*(x, u), v) in B

All the formation and introduction rules for the type \text{WORDS} (i.e. rules dealing with uninteresting membership subgoals) were connected to one rule \text{intro} which is easier to remember than all the individual rules. It is also used to expand the membership tactic to knowledge about \text{WORDS} which results in the very powerful tactic \text{member} where \text{member} uses \text{intro} 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.

\[
H, w : \text{WORDS}, H' \rightarrow T \quad \text{wordelim hyp}\\
\rightarrow T[c\langle w\rangle] \\
hd : \text{SYMBOLS}, tl : \text{WORDS}, tlhyp : T[tl\langle w\rangle] \rightarrow T[hd, tl\langle w\rangle]
\]

\[
H, w : \text{WORDS}, H' \rightarrow T \quad \text{wordelim_tail hyp}\\
u : \text{WORDS} \rightarrow T[u\langle w\rangle] \text{ in U10} \\
\rightarrow T[c\langle w\rangle] \\
a : \text{SYMBOLS}, v : \text{WORDS},Tv\langle w\rangle \rightarrow T[v\langle a\langle w\rangle\rangle]
\]

\[
H, w : \text{WORDS}, H' \rightarrow T \quad \text{wordelim_lg hyp}\\
u : \text{WORDS} \rightarrow T[u\langle w\rangle] \text{ in U10} \\
\rightarrow T[c\langle w\rangle] \\
ii : \text{int}, 0 < i, (\forall v : \text{WORDS}. |v| = i - 1 \text{ in int} \rightarrow T[v\langle w\rangle]), u : \text{WORDS}, |u| = i \text{ in int} \\
\rightarrow T[u\langle w\rangle]
\]

The latter rules were written as tactics using the Nuprl-theorems \text{ind principle} and \text{ind principle1}.

• \text{ind principle}
  THM \rightarrow \forall p : \text{WORDS} \rightarrow \text{U10}. (p (\varepsilon) \& \forall a : \text{SYMBOLS}. \forall z : \text{WORDS}. p(z) \rightarrow p(z + b)) \Rightarrow \forall w : \text{WORDS}. p(w)

• \text{ind principle1}
  THM \rightarrow \forall p : \text{WORDS} \rightarrow \text{U10} \\
  (p (\varepsilon) \& \forall i : \text{int}. 0 < i \rightarrow (\forall z : \text{WORDS}. |z| = i - 1 \text{ in int} \rightarrow p(z)) \Rightarrow (\forall z : \text{WORDS}. |z| = i \text{ in int} \rightarrow p(z))) \\
  \Rightarrow \forall w : \text{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 \text{U10}. Therefore \text{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.

.19.
resulting from it which describe the basic laws of operations on strings. They cover trivial properties like concatenation with the empty word

* _\texttt{eps}2_  
  \texttt{THM} \Rightarrow \forall v : \texttt{WORDS}. \ v, \varepsilon = v \text{ in } \texttt{WORDS} 
which yields to the rule

\[
H \Rightarrow v, \varepsilon = w \text{ in } \texttt{words} \text{ by } \texttt{wreduce 1} \\
H \Rightarrow v = w \text{ in } \texttt{words} \text{ by } \texttt{eps_concat_right 1}
\]

or the law of associativity of concatenation

* _\texttt{con_assoz}_  
  \texttt{THM} \Rightarrow \forall u, v, w : \texttt{WORDS}. \ ((u \cdot v) \cdot w) = (u \cdot (v \cdot w)) \text{ in } \texttt{WORDS} 

\[
H \Rightarrow (u \cdot v) \cdot w = w \text{ in } \texttt{words} \text{ by } \texttt{wreduce 1} \\
H \Rightarrow u \cdot (v \cdot w) = w \text{ in } \texttt{words} \\
H \Rightarrow \forall u \text{ in } \texttt{words} \\
H \Rightarrow v \text{ in } \texttt{words} \\
H \Rightarrow w \text{ in } \texttt{words} \text{ by } \texttt{con_assoz 1}
\]

the reverse of a concatenated word

* _\texttt{rev_con}_  
  \texttt{THM} \Rightarrow \forall u, v : \texttt{WORDS}. \ (+((u \cdot v))) = (+v \cdot u) \text{ in } \texttt{WORDS} 

\[
H \Rightarrow +((u \cdot v)) = w \text{ in } \texttt{words} \text{ by } \texttt{wreduce 1} \\
H \Rightarrow +v \cdot u = w \text{ in } \texttt{words} \\
H \Rightarrow u \text{ in } \texttt{words} \\
H \Rightarrow v \text{ in } \texttt{words} \text{ by } \texttt{rev_con 1}
\]

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

* _\texttt{lgprop1}_  
  \texttt{THM} \Rightarrow \forall w : \texttt{WORDS}. \ w \varepsilon \Leftrightarrow 0 < \lvert w \rvert 

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

* _\texttt{rangethm}_  
  \texttt{THM} \Rightarrow \forall w : \texttt{WORDS}. \ \forall l, r : \text{int}. \ 0 \leq l \ & \ 1 \leq r \ & \ r \leq \lvert w \rvert \Rightarrow \ w = (w[l \ldots l] \cdot w[l+1 \ldots r]) \cdot w[r+1 \ldots l] \text{ in } \texttt{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. _\texttt{rangethm}_). Like the introduction rules all the computation rules for \texttt{WORDS} were connected into a single rule _\texttt{wreduce}_. Thus the individual rulenames become unimportant for a user unless he intends to write efficient tactics.

The tactics written for \texttt{WORDS} deal with all-introduction (e.g. _\texttt{word_all_intro}_), induction on \texttt{WORDS} (i.e. all-introduction and then elimination of the introduced word), repeated introduction of words for existential formulas (_\texttt{word_some_intro}_) and two special cases as described in appendix B under \texttt{words_tactics.m1}.
4. Deterministic finite Automata

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

\[
\text{STATES} \rightarrow \text{FINITE SETS} \\
\text{DFA} \rightarrow (Q:\text{STATES} \rightarrow (\{(Q)\#\text{SYMBOLS}\} \rightarrow Q) \# Q \# P((Q)))
\]

\[
<\text{del}:Q\#\text{SYMBOLS} \rightarrow \{Q\}^{*} > \\
<\text{word}:\text{WORDS} \in L(\langle \text{AUT}\#\text{DFA} \rangle) > \\
<\text{AUT}:\text{DFA} > \\
L(<\text{AUT}:\text{DFA} >)
\]

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**

  \[
  \text{THM} \gg \forall M\#\text{DFA}. \exists n : \text{int}. \forall z : \text{WORDS}. z \in L(M) & n \mid z \\
  \rightarrow \exists u,v,w : \text{WORDS} . \quad z = (u*v)^n w \in L(M) \quad |u*v| \leq n \& 0 < |v| \& \forall k : \text{int}. (u*(v*k)) \in L(M)
  \]

I preferred the formulation \( \forall z : \text{WORDS}. z \in L(M) \rightarrow \exists n \mid z \) to \( \forall z : L(M) \) because otherwise the property \( d^*(Q0,z) = \{f\} \) -where \( (Q,d,q0,F) = M\)- would stay hidden in the proof too long. This shows one of the strong disadvantages 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 separate lemma. The reason was that several versions of it occured as subgoals and that there was no way to avoid this. The tactic \text{THEOREM} now could take care of this problem which was rather PRL-specific.

- **pumping_type**

  \[
  \text{THM} \gg \forall M\#\text{DFA}. \forall n : \text{int}. \forall z : \text{WORDS}. \forall u,v,w : \text{WORDS} . \quad (z = (u*v)^n w \in L(M) \& |u*v| \leq n \& 0 < |v| \& \forall k : \text{int}. (u*(v*k)) \in L(M)) \text{ in UI}
  \]

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 \texttt{dfa_intro_elim} which also takes care of wellformedness problems (e.g. \texttt{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 \texttt{pumping_type}.

```
+ EDIT THM pumping
+-------------------------------
+ * top
+ >> \forall M: \textsc{DFA}. \exists n: \textsc{int}. \forall z: \textsc{WORDS}. z \in L(M) \& n \equiv |z|
+ 4. \Rightarrow \exists u,v,w: \textsc{WORDS}
+ 5. \quad z = ((u+v)^*w) \in \textsc{WORDS} \& |(u+v)| \equiv n \& 0 < |v| \& \forall k: \textsc{int}. (|(u+(v^k))\cdot w|) 
+ \in L(M)
+ BY (* Let \( M = (Q,d,q_0,F) \) be a deterministic finite automaton *) \texttt{dfa_intro_elim}
+ 1* 1. M: \textsc{DFA}
+ 2. Q: \textsc{STATES}
+ 3. d: ((Q \cdot \#\textsc{SYMBOLS}) \rightarrow Q)
+ 4. q_0: Q
+ 5. F: P(Q)
+ 6. (Q,d,q_0,F) = M in \textsc{DFA}
+ >> \exists n: \textsc{int}. \forall z: \textsc{WORDS}. z \in L(Q,d,q_0,F) \& n \equiv |z|
+ 4. \Rightarrow \exists u,v,w: \textsc{WORDS}
+ 5. \quad z = ((u+v)^*w) \in \textsc{WORDS} \& |(u+v)| \equiv n \& 0 < |v| \& \forall k: \textsc{int}. (|(u+(v^k))\cdot w|) 
+ \in L(Q,d,q_0,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
+  \[ \text{word} = \text{equality} \]
+  \[ \text{Intro THENL} \]
+  \[ \text{- accepted_intro ; Wmember} \]
+  \[ \text{Intro THENL} \]
+  \[ \text{wqual}; \text{Intro THENL} [\text{wqual}; \text{Intro THENL}[\text{wqual}; \text{THEOREM pumping_type}]]\]
+ ]]
+ 1* >> \forall z: \textsc{WORDS}. z \in L(Q,d,q_0,F) \& (Q) \equiv |z|
+ 4. \Rightarrow \exists u,v,w: \textsc{WORDS}
+ 5. \quad z = ((u+v)^*w) \in \textsc{WORDS} \& |(u+v)| \equiv (Q) \& 0 < |v| \& \forall k: \textsc{int}. (|(u+(v^k))\cdot w|) 
+ \in L(Q,d,q_0,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 $z \in L(Q \cdot d \cdot q_0 \cdot F) \& (Q) = |z|$ will be found after the introduction steps. The tactic `membership` together with a `dfa-specific` tactic deals with the wellformedness subgoals.

Since $z$ is supposed to be longer than $n$, there must be a loop in the transition diagram, i.e. some $i < j \leq n$ such that $d^*(q_0, z[1..i]) = d^*(q_0, (z[1..i] + z[i+1..j])$). This fact is sequenced in and unraveled for further use. Two subgoals are to prove now.
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

\[ k: \mathbb{N}, \ l: \mathbb{N}, \ 0 \leq k, \ k \leq l, \ 1 \leq n, \ d^*(q_0, z[1..k]) = d^*(q_0, z[1..l]) \text{ in } Q \]

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

```plaintext
EDIT THM Pumping

* top 1 1 1
1. M: DFA
2. Q: STATES
3. d:(Q\#SYMBOLS)->Q
4. q_0: Q
5. F: P(Q)
6. (Q,d,q_0,F)->M in DFA
7. z: WORDS
8. z in (Q,d,q_0,F)
9. (Q)z in [z]

\[ \exists i:j: \mathbb{N} \& i < j \& j \leq Q \& d^*(q_0, z[1..i]) = d^*(q_0, (z[1..i] \rightarrow z[i+1..j])) \text{ in } Q \]

BY (* apply the pigeonhole principle *)

(-THEOREM1 'pigeon_variant' '[Q'; '(\n.d*(q_0,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; ITDAC; Equal; Equal; -repeat_and_intro])

THENL
[intro
THENL [membership; IntroTHENL [-state_set_intro; delstar_apply; delstar_apply]]
; -SubsToFor '[z[1..k] \rightarrow z[k+1..l]] = z[1..l] \text{ in } WORDS'
THENL
[THEOREM range9; Intro THENL [-state_set_intro; delstar_apply; delstar_apply]]
]
```

-24-
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 $w = z[j+1..l]$ 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 \times \text{SYMBOLS}) \rightarrow Q
4. q_0 : Q
5. F : P(Q)
6. (Q,d,q_0,F) = M in DFA
7. z : \text{WORDS}
8. z \in L(Q,d,q_0,F)
9. (Q) \vdash z
10. i : int
11. j : int
12. d^*(q_0, z[1..i]) = d^*(q_0, (z[1..i] * z[i+1..j])) in (Q)
13. 0 \leq i
14. i < j
15. j > 0
>> z = ((z[1..i] * z[i+1..j]) * z[j+1..l]) in \text{WORDS}
```

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

```
EDIT THM pumping
* top 1 1 1 2 1
1. M : DFA
2. Q : STATES
3. d : (Q \times \text{SYMBOLS}) \rightarrow Q
4. q_0 : Q
5. F : P(Q)
6. (Q,d,q_0,F) = M in DFA
7. z : \text{WORDS}
8. z \in L(Q,d,q_0,F)
9. (Q) \vdash z
10. i : int
11. j : int
12. d^*(q_0, z[1..i]) = d^*(q_0, (z[1..i] * z[i+1..j])) in (Q)
13. 0 \leq i
14. i < j
15. j > 0
>> z = ((z[1..i] * z[i+1..j]) * z[j+1..l]) in \text{WORDS}
```

```
BY (* Choose u = z[1..i], v = z[i+1..j], w = z[j+1..l] *)
word_some_intro 1 ['z[1..i]'); ('z[i+1..j]'); ('z[j+1..l]')
THENL [ THEOREM pumping_type, repeat_and_intro ]
1* >> z = ((z[1..i] * z[i+1..j]) * z[j+1..l]) in \text{WORDS}
2* >> |(z[1..i] * z[i+1..j])| = (Q)
3* >> 0 \leq |z[i+1..j]|
4* >> \forall k : int. ((z[1..i] * (z[i+1..j] * k)) * z[j+1..l]) \epsilon L(Q,d,q_0,F)

```

```
THEOREM range_thm
```
EDIT THM pumping

* top 1 1 1 2 2
1. M:<DFA
2. Q:STATES
3. d:((Q)xSYMBOLS)-->(Q)
4. q0:Q
5. F:=(p:Q)
6. (Q,d,q0,F)=M in DFA
7. W:WORDS
8. z (Q,d,q0,F)
9. (Q)=z
10. i:int
11. j:int
12. d*(q0,z[..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| - 1 < j )

* SubstFor '[z[i+1..j]]=j-i in int'
THEOREM range; THEOREM lgz; THEN &Arith; Wmember]
In the fourth subgoal the definition of \( z \in L(M) \) is transformed into its original form \( d^*(q_0, z) \in (F) \) and the same is done to the conclusion after the introduction of \( k \).

The proof follows immediately if \( d^*(q_0, u^*(v^k)^*w) - d^*(q_0, u^*v^w) = d^*(q_0, z) \) can be shown. This again can be reduced to the problem \( d^*(d^*(q_0, u), (v^k)) = d^*(q_0, u^v) \) 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 TM p m

* top 0 0 1 2 4 1 1
1. Q:STATES
2. d(Q #SYMBOLS)->(Q)
3. q0:
4. z:W R S
5. i:int
6. j:int
7. d*(q0,z[1..j]) - d*(q0,(z[1..i]+z[i+1..j])) in (Q)
8. i:
9. k:int
>> d*(d*(q0,z[i+1..j]),(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* = c are
trivial because they follow directly from hypotheses 7 *)

Int m
THEN
[ SubstFor 'z[i+1..j]*m = c in WORDS'
THENL
| -Iter_down 1 THEN weps
| -Star_reduction_eps 1
| ; Cumulativity THEN Intro THENL [-state_set_intro; delstar_apply; delstar_apply]
1 10 omap_apply_arg2 1 THEN -star_reduction_eps 1

>> d*(d*(q0,z..i),z..i+m -
  d*(d*(q0, [1 .. j]), ( [1 .. j] (-1))) - d*(q0,(z[1..i]+z[i+1..j])) in (Q)
( q0 [1 j] (z[1 .. j]) ) - d*(q0,(z[1..i]+z[i+1..j])) in (Q)

EDIT TM p m

* top 1 1
1. Q:STATE------------ - -
2. d(Q #SYMBOLS)->(Q)
3. q0:
4. z:W R S
5. i:int
6. j:int
7. d*(q0,z)
8. i:
9. k:int
10. m:int
11. d*(q0, [1 .. i]) = d*(q0,(z[1..i]+z[i+1..j])) in (Q)
12. hyp: d*(d*(q0,z[i+1..j]),(z[i+1..j]+(m-1))) = d*(q0,(z[1..i]+z[i+1..j])) in (Q)
>> d*(d*(q0,z[i+1..j]),(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* )
  d*(d*(q0,u),v*(v*k-1)) - ( properties of delstar )
  d*(d*(q0,u),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
| ; Cumulativity THEN Intro THENL [-state_set_intro; delstar_apply; delstar_apply]
| ; Cumulativity THEN Intro THENL [-state_set_intro; delstar_apply; delstar_apply]
| ;Cumulativity THEN Intro THENL [-state_set_intro; delstar_apply; delstar_apply]
...
Proof-tree of the Pumping Lemma

Let $M = (Q,d,q_0,F)$ be a deterministic finite automaton

Choose $n$ to be the number of states in $Q$

Let $z$ be a string such that $z \in L(M)$ and $n \leq |z|$

Since $z$ is longer than $n$ there must be a loop in the transition diagram i.e. some $i \leq n$ such that $d^*(q_0,z[i..i]) = d^*(q_0,z[i+1..i])$

Prove this claim by an application of the pigeonhole principle

Choose $u = z[1..i], v = z[i+1..j], w = z[j+1..|z|]$

Split goal into its single parts

Clear by definition of $uv$ and $w$ (Apply "range theorem")

$|uv| = j \in Q$

$|v| = j - i > 0$

Let $k$ be an integer. Go back to the definition of $z \in L(M)$ resp. $u \in v \in w \in L(M)$

Show:

$d^*(q_0, uv \uparrow k \downarrow w) = d^*(q_0, uv \uparrow k \downarrow w)$

which can be reduced to:

$d^*(d^*(q_0, u \uparrow k \downarrow v), v \downarrow k \downarrow w) - d^*(q_0, uv)$

Proof by induction on $k$

The down and base cases, where $v \uparrow k = \varepsilon$ are trivial

Do the final substitutions

$d^*(d^*(q_0, u \uparrow k \downarrow v), v \downarrow k \downarrow w)$

(by definition of $v \uparrow k$)

$d^*(d^*(q_0, u \uparrow k \downarrow v), v \downarrow k \downarrow w)$

(by properties of $d^*$)

$d^*(d^*(q_0, u \uparrow k \downarrow v), v \downarrow k \downarrow w)$

(by properties of $d^*$)

$d^*(d^*(q_0, u \uparrow k \downarrow v), v \downarrow k \downarrow w)$

(by induction hypothesis)
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 $z \in L(M)$ 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, λa.q.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 ))
     ;
(* Automaton for 2(112)* *)
let M1 = (5,λa.q.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))
      ;
(* 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.e );;
```

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.

```
NqPRL Command/Status
----------------------------------
E>pump(< M , 1,1,2,2,1,1.e >);;
  < 4 , < e ,< 1,1,2,2.e , 1,1.e > >
E>pump(< M , 2,2,1,1,2,2,2,2,2,1,1.e >);;
  < 4 , < e ,< 2,2,1,1.e , 2,2,2,2,2,1,1.e > >
E>pump(< M1 , 2,1,1,2,1,1,2.e >);;
  < 5 , < 2,1.e , < 1,2,1.e , 1,2.e > >
E>pump(< M1 , 2,1,1,2,1,1,2,1,1,2,1,1,2.e >);;
  < 5 , < 2,1.e , < 1,2,1.e , 1,2,1,1,2,1,1,2,1,1,2.e > >
```

-30-
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.

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

\[
\text{NFA} = (Q: \text{STATES} \# (\{(Q\#\text{SYMBOLS}):-P(Q)) \# (Q \# P(Q)))}
\]

\[
\text{d:Q\#\text{SYMBOLS}\rightarrow P(Q)\#}
\]

\[
\text{<w:WORDS> e L(<A: NFA>) = TRK( \lambda q, \{q:\{Q\}\}, \lambda f, a, \{p:Q|| \exists r: f. p e (d^*(q, a)) \})}
\]

\[
\text{L(<A: NFA>) = TRK( \lambda q, \{q:\{Q\}\}, \lambda f, a, \{p:Q|| \exists r: f. p e (d^*(q, a)) \})}
\]

All the rules dealing with non-deterministic 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

\[
\gg \text{w: NFA, } \exists \text{a: DFA, L(M) = L(A) in TYPE}
\]

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 \( \{1, \ldots, 2^*Q \} \) could be used for that where \( 2^*Q \) stands for the exponentiation of \( Q \).

\[
\begin{align*}
\text{2^*<i:int>} &= \text{ind}(<i>; x, y, 1; 1; x, \text{two_exp_i}, 2^\text{two_exp_i}) \\
\text{no(Q:STATES),<A:P(Q)> into \{1, \ldots, 2^*Q\}} &= \text{sum(2^*<i:intA>)} \\
\text{set(Q:STATES),<n:1, \ldots, 2^*Q> into P(Q))} &= \{ q; (Q) || (2^*q-1 < n \mod (2^*(2^*q))) \\
\text{d^*(Q:STATES),<A,P(Q),<w:WORDS>)} &= \{ q; (Q) || \exists r: (A). q e (d^*(r, w)) \}
\end{align*}
\]

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

1. Let \( M = (Q, d, q_0, F) \) be a nfa
2. Choose \( A = (Q_1, d_1, q_1, F_1) = (2^*Q, \lambda q, a, \text{no}(d^*(Q, \text{set}(Q, q), a)), \text{no}(Q_0, Q), \{ f; (2^*Q) || \exists g: (\text{set}(Q, f)). g e (F) \}) \)
   Show that \( A \) is in \( \text{DFA} \) and that the rest of the expression is well-formed
3. Show \( \forall w: \text{WORDS}. \exists p: Q. d^*(p, w) = \text{no}(d^*(Q, p, w)) \) by tail-induction on \( w \)
4. Reduce \( L(M) = L(A) \) in \( \text{TYPE} \) using computation rules to
   \( \exists r: (d^*(Q, w)), f e (F) \implies d^*(w, q_1, w) \in \{ f; (2^*Q) \mid \exists g: (\text{set}(Q, f)). g e (F) \}
5. Proof this by the following substitution sequence
   \( \exists g: (\text{set}(Q, f)). g e (F) \)
   \( \implies \text{no}(d^*(Q, q, w)) \in \{ f; (2^*Q) \mid \exists g: (\text{set}(Q, f)). g e (F) \}
   \implies \text{no}(d^*(Q, q, w)) \in \{ f; (2^*Q) \mid \exists g: (\text{set}(Q, f)). g e (F) \}
   \implies \exists g: (\text{set}(Q, q, w)). g e (F)
   \implies \exists g: (\text{set}(Q, q, w)). g e (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 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 wε) which does not occur in the original proof is necessary. Definitions also
become more complicated.

\[
\begin{align*}
\text{EPS} & \quad \text{⇒} (w:\text{WORDS} \quad w=\varepsilon \text{ in } \text{WORDS}) \\
\varepsilon\text{-NFA} & \quad \text{⇒} (Q:\text{STATES} \quad ((Q)@(\text{SYMBOLS}|\text{EPS})\rightarrow P((Q)) \quad \# \quad (Q) \quad P((Q))) \\
\varepsilon\text{-cl}(\langle M:\varepsilon\text{-NFA}, \langle n:N, \langle q:(Q) \rangle \rangle) & \quad \text{⇒ ind}(\langle n \rangle; m, \varepsilon\text{-cl}(\langle q:(Q) \rangle; (q:(Q))); \\
m, \varepsilon\text{-cl}(\langle p:(Q) \rangle; \lambda r: (\text{cl}). p @ (d(r, \text{inl}(\varepsilon))) \quad \text{where } (Q, d, q, 0, F)=M-) \\
\varepsilon\text{-closure}(\langle M: \varepsilon\text{-NFA}, \langle q:(Q) \rangle \rangle) & \quad \text{⇒} (p:(Q) | \\
\varepsilon\text{-closure}(\langle M: \varepsilon\text{-NFA}, \langle A: P((Q)) \rangle) & \quad \text{⇒} (p:(Q) | \\
\langle d:Q\times\text{SYMBOLS} \rightarrow P(Q) \rangle^* [\langle M: \varepsilon\text{-NFA} \rangle] & \quad \text{⇒} \text{TRK}(\langle q, \varepsilon\text{-closure}(\langle M, q \rangle), \lambda f, a. (p:(Q) \\
\langle w:\text{WORDS} \rangle \varepsilon L(\langle A: \text{NFA} \rangle) & \quad \text{⇒} (f) \quad \text{where } (Q, d, q, 0, F)\text{-}\langle A \rangle-
\langle w, \text{WORDS} \rangle \varepsilon L(\langle A \rangle))
\end{align*}
\]

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. \text{WORDS}) would have been added to PRL. This however is under user control
and subject to improvements.

Some of the extensions described above (\text{general tactics}, \text{logic}, \text{WORDS}) might be used the way they
are (maybe some additions should be made) while others (sets, \text{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>

* all1
  DEF ∀<var>,<var>:<type>. <prop>

* all2
  DEF ∀<var>,<var>:<type>. <prop>

* all3
  DEF ∀<var>,<var>,<var>:<type>. <prop>

* some
  DEF ∃<var>:<type>. <prop>

* some1
  DEF ∃<var>,<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 \( \lambda id,id,term \)

c12
DEF
1.2: SETS

-----
c12a
DEF
1.2.a SETS & POWERSETS

-----
sets
DEF SETS

setdef
DEF (\{ \langle id, <TYPE>|<TYPE> \} )

singleton
DEF (\{ \langle element, <TYPE> \} )

carr
DEF carrier(SETS)

pred
DEF pred(SETS)

inspect
DEF \langle element \rangle \in (SETS)

setconversion
DEF (SETS into a set)

powerset
DEF P(<TYPE>)

# set1
THM \forall A, <TYPE>. \forall S: P(A). \exists p:A \rightarrow \exists U. S = (A, p) in SETS

# set3
THM \forall S: SETS. \forall x: carrier(S). pred(S)(x) \leftrightarrow x \in S

# set4
THM \forall A, <TYPE>. \forall S: P(A). \forall x, S \in A

---
c12b
DEF
1.2.b NATURAL NUMBERS & NUMBER THEORY

-----

leq
DEF \langle int \rangle \leq \langle int \rangle

uneq
DEF \langle int \rangle \neq \langle int \rangle

N
DEF N

int_arith
THM \forall x, y: \langle int \rangle. (x = y in int) | x\#y

intsub
THM \forall i, j: \langle int \rangle. 1-j in \langle int \rangle \rightarrow j-i = 0 in \langle int \rangle

less_arith
THM \forall i, j: \langle int \rangle. i < j \rightarrow 0 < j-i

less_arith1
THM \forall i, j, k: \langle int \rangle. i \leq j \& 0 < k \Rightarrow i \leq j+k
\* cl2c
\* 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 \forall A;FINITE SETS. (A) in TYPE

\* wf1
\* THM \forall n:N. \forall x:[1...,n]. x in \text{int}

\* wf2
\* THM \forall n:N. \forall x:[1...,n]. x \leq n

\* wf3
\* THM \forall n:N. \forall x:[1...,n]. 0 < x

\* finiteq
\* THM \forall m:N. \forall x:[1...,m-1]. x in [1...,m]

\* extend
\* THM \forall k:N. \forall y:[1...,k]. y in [1...,k+1]

\* extend1
\* THM \forall k:N. \forall x:[1...,k+1]. x in [1...,k+1]

\* restrict
\* THM \forall m:N. \forall x:[1...,m]. x < m \Rightarrow x in [1...,m-1]

\* restrict1
\* THM \forall m:N. \forall x:[1...,m]. x = m \Rightarrow x in [1...,m-1]

\* finarg
\* THM \forall m:N. \forall x:[1...,m]. (x = m \in [1...,m]) \mid (x < m)

\* finfuntype
\* THM \forall m:N. \forall f:[1...,m+1] \rightarrow [1...,m]. \forall x:[1...,m+1]. f(x) \in \text{int}

\* hole1
\* THM \forall k:int. 0 \leq k \Rightarrow \forall n:N. \forall f:[1...,k] \rightarrow [1...,n]. \forall y:[1...,n].
\quad \exists x:[1...,k]. f(x) \neq y \in \text{int} \mid \forall x:[1...,k]. f(x) \neq y

\* pigeon-type
\* THM \forall k:N. \forall f:[1...,k+1] \rightarrow [1...,k]. \forall i,j:[1...,k+1].
\quad (i \neq j \land f(i) = f(j) \text{ in int}) \Rightarrow \text{in U}

\* pigeon
\* THM \forall k:int. 0 \leq k \Rightarrow \forall f:[1...,k+1] \rightarrow [1...,k]
\quad \exists i,j:[1...,k+1]. i \neq j \land f(i) = f(j) \text{ in int}

\* pigeon-variant
\* THM \forall n:N. \forall h:[0...,n] \rightarrow [1...,n]
\quad \exists k: \text{int}
\quad h(k) = h(1) \text{ 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. (ε∗w) = a.w in WORDS
* symcon
  THM □∀u,v:WORDS. ∀a:SYMBOLS. (a.u+v) = a.(u+v) in WORDS
* conasso2
  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 \( \forall a : \text{SYMBOLS}. \ \forall u : \text{WORDS}. \ + a u = +u a \ \text{in WORDS} \)

* anticonsrev
  THM \( \forall a : \text{SYMBOLS}. \ \forall u : \text{WORDS}. \ + a u = +u a \ \text{in WORDS} \)

* revcon
  THM \( \forall u, v : \text{WORDS}. \ ((u v)) = (v u) \ \text{in WORDS} \)

* doublerev
  THM \( \forall u : \text{WORDS}. \ (++ u) = \text{u in WORDS} \)

* hdprop
  THM \( \forall a : \text{SYMBOLS}. \ \forall u : \text{WORDS}. \ \text{hd}(a u) = a \ \text{in SYMBOLS} \)

* hd1
  THM \( \forall u : \text{WORDS}. \ u \in \epsilon \Rightarrow \text{hd}(u v) = \text{hd}(u) \ \text{in SYMBOLS} \)

* t1prop
  THM \( \forall a : \text{SYMBOLS}. \ \forall u : \text{WORDS}. \ \text{tl}(a u) = u \ \text{in WORDS} \)

* t11
  THM \( \forall a : \text{SYMBOLS}. \ \forall u, v : \text{WORDS}. \ \text{tl}((a u v)) = (u v) \ \text{in WORDS} \)

* t1con
  THM \( \forall u, v : \text{WORDS}. \ u \in \epsilon \Rightarrow \text{tl}(u v) = (\text{tl}(u) v) \ \text{in WORDS} \)

* lgtype1
  THM \( \forall w : \text{WORDS}. \ \epsilon | w | \)

* lgprop
  THM \( \forall w : \text{WORDS}. \ w \in \epsilon \ \text{in WORDS} 
    \quad \Rightarrow | w | = 0 \ \text{in int} \)

* lgprop1
  THM \( \forall w : \text{WORDS}. \ w \in \epsilon \ \text{in WORDS} 
    \quad \Rightarrow 0 < | w | \)

* lgps
  THM \( | \epsilon | = 0 \ \text{in int} \)

* lgcons
  THM \( \forall a : \text{SYMBOLS}. \ \forall u : \text{WORDS}. \ | a u | = (| u | + 1) \ \text{in int} \)

* lgbody
  THM \( \forall a : \text{SYMBOLS}. \ \epsilon | a | = 1 \ \text{in int} \)

* lgconcat
  THM \( \forall u, v : \text{WORDS}. \ |(u v)| = (| u | + | v |) \ \text{in int} \)

* lganti
  THM \( \forall w : \text{WORDS}. \ \forall a : \text{SYMBOLS}. \ w \in a \ \text{in WORDS} 
    \quad \Rightarrow |\text{w}a\| = |\text{w}| + 1 \ \text{in int} \)

* lgreq
  THM \( \forall w : \text{WORDS}. \ |+w| = |w| \ \text{in int} \)

* lglt1
  THM \( \forall w : \text{WORDS}. \ w \in \epsilon \Rightarrow |\text{tl}(w)| = (|w| - 1) \ \text{in int} \)

* lgpre
  THM \( \forall w : \text{WORDS}. \ \forall k : \text{int}. \ \text{0} \leq k \ \&\ \text{0} \leq | w | \Rightarrow | w[k+1..|w|] | = | w | - k \ \text{in int} \)

* lgsel1
  THM \( \forall w : \text{WORDS}. \ \forall i : \text{int}. \ \text{0} \leq i \Rightarrow | w[i..1] | = 1 \ \text{in int} \)

* lgselu
  THM \( \forall w : \text{WORDS}. \ \forall i : \text{int}. \ \text{0} \leq | w[i..] | = 1 \ \text{in int} \)

* lgrange
  THM \( \forall w : \text{WORDS}. \ \forall i, r : \text{int}. \ \text{0} \leq i \ \&\ \text{0} \leq r \Rightarrow | w[i+1..r] | = r - i \ \text{in int} \)

* lter1
  THM \( \forall w : \text{WORDS}. \ \forall k : \text{int}. \ k \in w \Rightarrow \text{w}k \in \epsilon \ \text{in WORDS} \)

* lter2
  THM \( \forall w : \text{WORDS}. \ \forall k : \text{int}. \ \text{0} < k \Rightarrow \text{w}k = \text{w}w\text{rk} \ \text{in WORDS} \)

* word1
  THM \( \forall w : \text{WORDS}. \ \forall a : \text{SYMBOLS}. \ \exists z : \text{WORDS}. \ \exists b : \text{SYMBOLS}. \ a \cdot l = z \cdot b \ \text{in WORDS} \)

* notepsl
  THM \( \forall w : \text{WORDS}. \ (w = \epsilon \ \text{in WORDS}) \ | w \epsilon \)

* notepsthm
  THM \( \forall w : \text{WORDS}. \ w \epsilon \Rightarrow \neg w \epsilon \ \text{in WORDS} \)
* preops
  THM >> \forall k: \text{int}. \ v[i+1..lg] = v in \text{WORDS}

* pre0
  THM >> \forall u: \text{WORDS}. \ \forall i: \text{int}. \ 0 \leq i \leq |u| \Rightarrow u[i+1..lg] = u

* pre1
  THM >> \forall w: \text{WORDS}. \ w[0+1..lg] = w in \text{WORDS}

* pre2
  THM >> \forall w: \text{WORDS}. \ \forall k: \text{N}. \ w[k+1..lg] = t1(w[k+1..lg]) in \text{WORDS}

* pre3
  THM >> \forall w: \text{WORDS}. \ \forall k: \text{int}. \ t1(w[k+1..lg]) = t1(w)[k+1..lg] in \text{WORDS}

* precons
  THM >> \forall a: \text{SYMBOLS}. \ \forall i: \text{WORDS}. \ \forall k: \text{int}. \ 0 < k
  \Rightarrow a.1[k+1..lg] = \text{t1}(k+1..lg) in \text{WORDS}

* pre3
  THM >> \forall w: \text{WORDS}. \ w[|w|+1..lg] = w in \text{WORDS}

* pre4
  THM >> \forall v: \text{WORDS}. \ \forall a: \text{SYMBOLS}. \ w*a[|w|+1..lg] = a in \text{WORDS}

* pre5
  THM >> \forall u, v: \text{WORDS}. \ \forall i: \text{int}. \ 0 \leq i \leq |u|
  \Rightarrow (u*v)[i+1..lg] = (u[i+1..lg] + v) in \text{WORDS}

# selrev
  THM >> \forall w: \text{WORDS}. \ \forall i: \{0, 1, \ldots, |w|\}. \ w[i] = w[|w|-i] in \text{SYMBOLS}

# selcon1
  THM >> \forall u, v: \text{WORDS}. \ \forall i: \text{int}. \ 0 < i \leq |u|
  \Rightarrow (u*v)[i] = u[i] in \text{SYMBOLS}

# selcon2
  THM >> \forall u, v: \text{WORDS}. \ \forall i: \text{int}. \ |u| < i \Rightarrow (u*v)[i] = v[i-|u|] in \text{SYMBOLS}

* sel1
  THM >> \forall w: \text{WORDS}. \ w(1) = \text{hd}(w) in \text{SYMBOLS}

# sel2
  THM >> \forall a: \text{SYMBOLS}. \ \forall w: \text{WORDS}. \ \forall i: \text{N}. \ a.w(i+1) = w(i) in \text{SYMBOLS}

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

# sel4
  THM >> \forall w: \text{WORDS}. \ \forall i: \{1, \ldots, |w|\}. \ (w(i)*w[i+1..lg]) = w[i-1+1..lg] in \text{WORDS}

# sel5
  THM >> \forall w: \text{WORDS}. \ \forall i, j: \text{N}. \ w[i+1..lg](j) = w(i+j) in \text{SYMBOLS}

* sel6
  THM >> \forall w: \text{WORDS}. \ \forall a: \text{SYMBOLS}. \ w*a(|w*a|) = a in \text{SYMBOLS}

* suf1
  THM >> \forall w: \text{WORDS}. \ w[1..0] = w in \text{WORDS}

* suf2
  THM >> \forall w: \text{WORDS}. \ \forall i: \text{N}. \ w[i+1..lg] = w[i+1lg] in \text{WORDS}

* sufcon
  THM >> \forall u, v: \text{WORDS}. \ \forall k: \text{int}. \ k \leq |u|
  \Rightarrow (u+v)[1..k] = u[1..k] in \text{WORDS}

* suf3
  THM >> \forall w: \text{WORDS}. \ w[1..|w|] = w in \text{WORDS}

# suf4
  THM >> \forall w: \text{WORDS}. \ \forall i, j: \text{int}. \ 1 \leq i \Rightarrow w[1..lg](i) = w(i) in \text{SYMBOLS}

* suf5
  THM >> \forall u, v: \text{WORDS}. \ (u*v)[1..|w|] = w in \text{WORDS}

* range1
  THM >> \forall w: \text{WORDS}. \ \forall i: \text{int}. \ w[0+1..lg] = w[1..lg] in \text{WORDS}

* range2
  THM >> \forall w: \text{WORDS}. \ \forall i: \text{int}. \ 0 \leq i \leq |w| \Rightarrow w[i+1..lg] = w[i+1lg] in \text{WORDS}

# range3
  THM >> \forall w: \text{WORDS}. \ \forall i: \{1, \ldots, |w|\}. \ \forall r: \text{int}. \ 1 \leq r
  \Rightarrow w[1+1..lg] = w(i).w[1+1..lg] in \text{WORDS}
* range4
  \text{THM} \Rightarrow \forall w: \text{WORDS}. \forall l, r: \text{int}. \, 0 \leq l \land l < r
  \quad \Rightarrow \quad w[l..r] = w[l+1..r-1] * w(r) \text{ in WORDS}

* range5
  \text{THM} \Rightarrow \forall w: \text{WORDS}. \forall r: \text{int}. \, w[r+1..r] = \varepsilon \text{ in WORDS}

* range5a
  \text{THM} \Rightarrow \forall w: \text{WORDS}. \forall l, r, k: \text{int}. \, 0 \leq l \land l < r \land 0 \leq k
  \quad \Rightarrow \quad (w[l+1..r] * w[r+1..r+k]) = w[l+1..r+k] \text{ in WORDS}

* range6
  \text{THM} \Rightarrow \forall w: \text{WORDS}. \forall l, r, k: \text{int}. \, 0 \leq l \land l < r \land 0 \leq k
  \quad \Rightarrow \quad (w[l+1..r] * w[r+1..k]) = w[l+1..k] \text{ in WORDS}

* range7
  \text{THM} \Rightarrow \forall w: \text{WORDS}. \, w[0+1..|w|] = w \text{ in WORDS}

* range8
  \text{THM} \Rightarrow \forall w: \text{WORDS}. \forall l, r, i, n: \text{int}. \, 1 \leq r \Rightarrow \quad w[l+1..r](i) = w[l+i] \text{ in SYMBOLS}

* range9
  \text{THM} \Rightarrow \forall w: \text{WORDS}. \forall l, r: \text{int}. \, 0 \leq l \land l < r
  \quad \Rightarrow \quad (w[l+1..r]) = w[l..r] \text{ in WORDS}

* range10
  \text{THM} \Rightarrow \forall w: \text{WORDS}. \forall r: \text{int}. \, 0 \leq r \Rightarrow w[r+1..r] = w[r..r] \text{ in WORDS}

* c15
  \text{DEF}
  \begin{align*}
    & \text{Induction} \\
    & \text{induction_thm}
  \end{align*}

* inductive_thm
  \text{THM} \Rightarrow \forall r: \text{int}. \, \forall \alpha: \text{int} \Rightarrow w[r+1..r](\alpha) = w[r+1..r](\alpha)

* inductive_principle
  \text{THM} \Rightarrow \forall w: \text{WORDS} \Rightarrow \forall r: \text{int} \Rightarrow w[r+1..r] = w[r..r]

* inductive_thm1
  \text{THM} \Rightarrow \forall w: \text{WORDS} \Rightarrow \forall r: \text{int} \Rightarrow w[r+1..r] = w[r..r]

* inductive_principle1
  \text{THM} \Rightarrow \forall w: \text{WORDS} \Rightarrow \forall r: \text{int} \Rightarrow w[r+1..r] = w[r..r]

* inductive_principle2
  \text{THM} \Rightarrow \forall w: \text{WORDS} \Rightarrow \forall r: \text{int} \Rightarrow w[r+1..r] = w[r..r]

* trk
  \text{DEF} \quad \text{TRK}((A \Rightarrow B), (B \Rightarrow \text{SYMBOLS} \Rightarrow B))

* trktype
  \text{THM} \Rightarrow \forall A, B: \text{TYPE}. \text{forall:} A \Rightarrow B, \forall A: (B \Rightarrow \text{SYMBOLS} \Rightarrow B). \text{TRK}(g, h) \text{ in}\ (A \Rightarrow \text{WORDS}) \Rightarrow B

* trk1
  \text{THM} \Rightarrow \forall A, B: \text{TYPE}. \text{forall:} A \Rightarrow B, \forall A: (B \Rightarrow \text{SYMBOLS} \Rightarrow B). \text{forall:} A \Rightarrow B

* trk2
  \text{THM} \Rightarrow \forall A, B: \text{TYPE}. \text{forall:} A \Rightarrow B, \forall A: (B \Rightarrow \text{SYMBOLS} \Rightarrow B). \text{forall:} A \Rightarrow B

* idtrk
  \text{DEF} \quad \text{B} \Rightarrow \text{SYMBOLS} \Rightarrow B

* trk3
  \text{THM} \Rightarrow \forall A, B: \text{TYPE}. \text{forall:} A \Rightarrow B, \forall A: (B \Rightarrow \text{SYMBOLS} \Rightarrow B). \text{forall:} A \Rightarrow B

* trk4
CHAPTER 2:  FINITE AUTOMATA

2.1:  Basic definitions

* states
  DEF STATES
* dfa
  DEF DFA
* delstar
  DEF <QxSYMBOLS->Q>*
* delstartype
  THM ->Q:STATES. \forall d:(Q)xSYMBOLS ->(Q). d* in ((Q)xWORDS) -> (Q)
* delprop
  THM ->Q:STATES. \forall d:(Q)xSYMBOLS ->(Q). \forall q:(Q). d*(q,a) = q in (Q)
* delprop0
  THM ->Q:STATES. \forall d:(Q)xSYMBOLS ->(Q). \forall q:(Q). \forall w:WORDS. \forall a:SYMBOLS
  . d*(q,a*w) = d*(q,w) in (Q)
* delprop1
  THM ->Q:STATES. \forall d:(Q)xSYMBOLS ->(Q). \forall q,v,w:WORDS. \forall q:(Q)
  . d*(q,v*w) = d*(d*(q,v),w) in (Q)
* accepted
  DEF <WORDS> \in L(<DFA>)
* acceptedtype
  THM ->VM:DFA. \forall w:WORDS. w \in L(M) in UI
* accept
  DEF L(<DFA>)
* load-all
  ML

CHAPTER 3:  PROPERTIES OF REGULAR SETS

3.1:  PUMPING LEMMA

* pumping_type
  THM ->VM:DFA. \forall n:int. \forall z:WORDS. \forall u,v,w:WORDS
  . ( z=(u*v)*w ) in WORDS & |(u*v)|=n & 0 < |v|
  & \forall k:int. ((u*(v*k))*w) \in L(M)
  ) in UI
* pumping
  THM ->VM:DFA. \exists n:int. \forall z:WORDS. z \in L(M) & n=|z| \Rightarrow \exists u,v,w:WORDS
  . z=(u*v)*w ) in WORDS & |(u*v)|=n & 0 < |v|
  & \forall k:int. ((u*(v*k))*w) \in 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 instantiations refer to the PRL library "automata"
************ LIST OF USED PREDEFINED FUNCTIONS ************

map : (* -> **)#(*list) -> (**list)  
- : tok -> tok -> tok 
three_match : (term#term#tok list) -> term list

************ END LIST OF PREDEFINED FUNCTIONS *************

basic term definitions

INT : term  
NIL : term  
VOID : term  
avar : tok -> term  
imp : term  
andl : term  
andr : term  
zero : term  
one : term  
U1 : term  
U2 : term

elpfunctions

cat : tok -> int -> tok  
ls_member : * -> *list -> bool  
ids : declaration_list -> tok_list  
cut_from : int -> *list -> *list  
cut_first : int -> *list -> *list  

fs_alpha_convertible : term -> term -> bool

list : int -> * -> *list  
intlist : int -> int -> int list

first_exp : proof -> term  
ordered_equality : int -> proof -> (term#term#term)  

new : tok -> proof -> tok

TYPE 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 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 incomplete.

EXAMPLE:
  t = (x1:A1 -> A2 -> ( (x3:A3 -> B & B') # (y3:C3 ->C4) )
  returns a1.mp.and1.a3.andrl

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. mp, 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 :>pml> 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 **************'; '

; 'Elementary tactics & Rule-tactics

Intro = tactic
Set_elm = int -> int -> tactic

Hyp = int -> tactic
Last_hyp = tactic
Last_1_hyp = tactic
Last_2_hyp = tactic

CUMULATIVITY = int -> tactic
Cumulativity = tactic
Thinning = int_list -> tactic
Thin_last = tactic

Equal = tactic
Equal = tactic
Arith = tactic
Decision = tactic

Lemma = tok -> tactic
Seq = term list -> tactic

Extensionality = tactic

FAILTAC = tactic

Int_minus = tactic
Int_add = tactic
Int_sub = tactic
Int_mult = tactic
Int_number = tactic

Ui_equality = tactic
Ui_Ui = tactic
ProdUi = tactic
FuncUi = tactic

Int_computation = tok -> int -> tactic
Inteq_computation = int -> bool -> tactic

List_computation = int -> tactic

Int_elm = int -> tactic

Direct_computation = term -> tactic
Hyp_computation = int -> term -> tactic

Trivial_types = tactic

a collection of all the intro-rules which don't leave subgoal,
omputation tactics
-----------------------
H >> redex
  >> contractum
H >> redex in T
  >> contractum in T
H >> redex - t in T
  >> contractum = t in T
H >> t = redex in T
  >> t = contractum in T
H >> t in redex
  >> t in contractum
H >> y:redex -> T
  >> y:contractum -> T
H >> y:T -> redex
  >> y:T -> contractum
H >> t in y:T -> redex
  >> t in y:T -> contractum
H >> t in y:redex -> T
  >> 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)

Elementary tactuals
---------------------
THEMEMBER = 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

Hypothesis 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 a1 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 a1's
- nulltac will be used if inst_list is empty
- subgoals that the a1 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**

<table>
<thead>
<tr>
<th>Direct_Hypotheses</th>
<th>tactic</th>
<th>try all hypotheses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hyp_elim_on</td>
<td>tactic</td>
<td>same as eliminate using</td>
</tr>
<tr>
<td>HYP</td>
<td></td>
<td>try if goal follows from named hyp via function or product elimination (uses matching)</td>
</tr>
<tr>
<td>Try_elimination_in_hypotheses</td>
<td>tactic</td>
<td>try if goal follows from hypotheses via function or product elimination (uses matching)</td>
</tr>
</tbody>
</table>

**HYPOTHESIS**

tries all kinds of manipulations with hypotheses which end up without leaving subgoals. This includes decision procedures and elimination within hypotheses if they succeed.

**membership 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)

**THEOREM APPLICATION** (including elimination of \( \forall \) and \& if necessary ')

---------------

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

`THEOREM : tok -> tactic`

the named theorem be (as an example)

\[
\forall x_1 \exists x_2 . \forall x_3 . T \Rightarrow (( \forall x_4 : T_4 . T_5 ) \& ( S_4 \Rightarrow S_5 )) \& S_3
\]

THEOREM1 needs a given instantiation list and produces

\[
T \Rightarrow T_1 \& T_2 . (T_3 \Rightarrow (( \forall x_4 : T_4 . T_5 ) \& ( S_4 \Rightarrow S_5 ))) \& S_3
\]

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])

\[
T \Rightarrow T_1 \& T_2 . (T_3 \Rightarrow (( \forall x_4 : T_4 . T_5 ) \& ( S_4 \Rightarrow S_5 ))) \& S_3
\]

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.

`thm : tok -> tactic`

`th : tok -> tactic`

\( \exists \) and \& are defined as in the logic. For the implementation of THEOREM

...
Constructors, Destructors & Predicates

a: Constructors

- `make_and_term` = term → term → term
- `make_or_term` = term → term → term
- `make_imp_term` = term → term → term
- `make_equiv_term` = term → term → term
- `make_false_term/\text{FALSE}` = term

- `make_not_term` = term → term
- `make_all1_term` = term → term → term → term
- `make_all2_term` = term → term → term → term
- `make_all3_term` = term → term → term → term
- `make_some2_term` = term → term → term → term
- `make_some3_term` = term → term → term → term

p,q,type,term are placeholders for terms in \text{U1}, x,y,z for \text{var_terms}.

b: Destructors

Split the constructions named above into their parts. The result is a product term. E.g. `\text{destructure_all2}` = term → (term#term#term#term) : "\forall x,y: \text{type}, \text{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.

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

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

Ia: Formation

---

H >> A&B in U1 by intro
   >> A in U1
   >> B in U1

H >> A|B in U1 by intro
   >> A in U1
   >> B in U1

H >> A->B in U1 by intro
   >> A in U1
   >> B in U1

H >> A<->B in U1 by intro
   >> A in U1
   >> B in U1

H >> false in U1 by intro

H >> ~A in U1 by intro
   >> A in U1

H >> (\forall x:A.B) in U1 by intro
   >> A in U1
   x:A >> B in U1

H >> (\forall x1:A1......xn:An.B) in U1 by intro [n]
   >> A1 in U1
   x1:A1 >> A2 in U1
   ... x1:A1......xn:An >> B in U1

H >> (\exists x:A.B) in U1 by intro
   >> A in U1
   x:A >> B in U1

H >> (\exists x1:A1......xn:An.B) in U1 by intro [n]
   >> A1 in U1
   x1:A1 >> A2 in U1
   ... x1:A1......xn:An >> B in U1

---

and_equality

or_equality

imp_equality

equiv_equality

false_equality

not_equality

all_equality

repeat_all_equality

repeat_all_equality_for n

some_equality

repeat_some_equality

repeat_some_equality_for n

(--->> intro isn't written yet)
Ib: Introduction
-----------------

H \rightarrow A \cdot B by intro
  \rightarrow A
  \rightarrow B

H \rightarrow A \cdot B by intro left i
  \rightarrow A
  \rightarrow B \in \text{Ui}

H \rightarrow A \cdot B by intro right i
  \rightarrow A \in \text{Ui}

H \rightarrow A \rightarrow B by intro i
  \rightarrow A \rightarrow B
  \rightarrow A \in \text{Ui}

H \rightarrow A \leftrightarrow B by intro i
  \rightarrow A \leftrightarrow B
  \rightarrow A \in \text{Ui}

H \rightarrow false by intro

H \rightarrow \neg A by intro i
  \rightarrow A \rightarrow \text{FALSE}
  \rightarrow A \in \text{Ui}

H \rightarrow \forall x:A . B by intro i
  \rightarrow x:A \rightarrow B
  \rightarrow A \in \text{Ui}

H \rightarrow \forall x_1:A_1 , \ldots , x_n:A_n . B by intro i [n]
  \rightarrow x_1:A_1 , \ldots , x_n:A_n \rightarrow B
  \rightarrow x_1:A_1 , \ldots , x_{n-1}:A_{n-1} \rightarrow A_{n-1} \in \text{Ui}
  \rightarrow \ldots
  \rightarrow A_1 \in \text{Ui}

H \rightarrow \exists x:A . B by intro i a
  \rightarrow a \in A
  \rightarrow B[a/x]
  \rightarrow x:A \rightarrow B \in \text{Ui}

H \rightarrow \exists x_1:A_1 , \ldots , x_n:A_n . B by intro i a1..an [n]
  \rightarrow A_1 \in \text{Ui}
  \rightarrow x_1:A_1 \rightarrow A_2 \in \text{Ui}
  \rightarrow x_1:A_1 , \ldots , x_{n-1}:A_{n-1} \rightarrow B \in \text{Ui}
  \rightarrow a_1 \in A_1
  \rightarrow \ldots
  \rightarrow a_n \in A_n[a_1 , \ldots , a_{n-1}/x_1 , \ldots , x_{n-1}]
  \rightarrow B[a_1 , \ldots , a_n/x_1 , \ldots , x_n]


and_equality
or_intro_left level
or_intro_right level
imp_intro level
equiv_intro level
false_intro (H must be contradictory)
not_intro level
all_intro level
repeat_all_intro level
repeat_all_intro_for n level
some_intro level a
repeat_some_intro level alist
:ic: Elimination
-------------------

\[ H, A \& A_2 \ldots \& A_n, H' \Rightarrow T \]
\[ H, H', A_1, A_2, \ldots, A_n \Rightarrow T \]

\[ H, A \mid A_2 \mid \ldots \mid A_n, H' \Rightarrow T \]
\[ A_1 \Rightarrow T \]
\[ A_2 \Rightarrow T \]
\[ \ldots \]
\[ A_n \Rightarrow T \]

\[ H, A \Rightarrow B, H' \Rightarrow T \]
\[ \Rightarrow A \]
\[ B \Rightarrow T \]

\[ H, A \Leftarrow B, H' \Rightarrow T \]
\[ \Rightarrow B \]
\[ A \Rightarrow T \]

\[ H, \text{FALSE}, H' \Rightarrow T \]

\[ H, \text{NOT}, H' \Rightarrow T \]
\[ H, H' \Rightarrow A \]

\[ H, \forall x: A_1, \forall x_n: A_n, B, H' \Rightarrow T \]
\[ \Rightarrow a_1 \text{ in } A_1 \]
\[ \ldots \]
\[ \Rightarrow a_n \text{ in } A_n[a_1, \ldots, a_{n-1}, x_1, \ldots, x_{n-1}] \]
\[ B[a_1, \ldots, a_{n-1}, x_1, \ldots, x_n] \Rightarrow T \]

\[ H, \exists x_1: A_1, \ldots, \exists x_n: A_n, B, H' \Rightarrow T \]
\[ H, H', x_1: A_1, \ldots, x_n: A_n, B \Rightarrow T \]

:id: Computation
-----------------

\[ \text{jo computation for logic expressions} \]
II. Tactics

------------

H >> ∀x:A. B
  x:A >> B

by AllIntro j AUjtac

H >> ∀x1:A1,...x:n An. B
  x1:A1,..., x:n An >> B

by RepeatAllIntro j AIUjtacs

H >> ∃y:A B
   B[a/y]

by SomeIntro j a atac BUjtac

H >> ∃y1:A1,..., ∃yn:An. B
   B[a1,...,ay1,...yn]

by RepeatSomeIntro j a list ai atacs AUjtacs BUjtac

H >> A =⇒ B
   A >> B

by ImpIntro j AUjtac

H >> A ⇔ B
   A >> B
   B >> A

by EquivIntro j AUjtac BUjtac

ai atacs must prove "⇒ a i in Ai[a1,...a i-1/x1,...xi-1]"
AIUjtacs  "⇒ A i in Uj"
BUjtac    "⇒ B in Uj

Using IDTAC instead leaves the corresponding subgoals open

Short forms:
level j = 1
- All intro AUjtac
- Repeat_all_intro AIUjtacs
- Some_intro a atac BUjtac
- Repeat_some_intro a list ai atacs AUjtacs BUjtac
- Imp_intro AUjtac
- Equiv_intro AUjtac BUjtac

level j = 1 and AUjtac = membership (+BUjtac = membership)
- AllIntro
- ImpIntro
- EquivIntro

to be written:
- Repeat_and_equality
- Repeat_and_equality_for n
- Repeat_and_intro
- Repeat_and_intro_for n

I1-introduction for standard PRL-types

--------------

H >> ∀x:int. T
   x:int >> T

by int_all_intro

H >> ∀x:int. T
   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
Constructors, Destuctors & 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`  
  t a b c t'  into "t - where (a,b,c) = t'"

- `make_where4_term` = `term -> term -> term -> term -> term`  
  t a b c d t'  into "t - where (a,b,c,d) = t'"

**additional constructors**

- `make_id_function_term/fd` = `term`  
  t  into "\x.x"

- `make_lambda2_term` = `term -> term -> term`  
  x y t  into "\x.y t"

- `make_apply2_term` = `term -> term -> term`  
  f a b  into "f(a,b)"

**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`  
  >> a1 in A1

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

- `H > t - where (a,b) = x - in T`  
  where2_equality using

- `x in y:A#B`  
  a1:A, b1:B[a/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

- `a1:A, b1:A2[a/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

- `y1:A1 >> y2:A2#y3:A3#A4 in Ui`  
  y1:A1, b1:A2[a/y1], c1:A3[a1,b1/y1,y2], d1:A4[a1,b1,c1/y1,y2,y3]  
  >> t[a1,b1,c1,d1] in y1:A1#y2:A2#y3:A3#A4

- `H > \lambda a,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

\[ \text{Id: Computation} \]
\[
\text{\begin{align*}
\text{H} \Rightarrow t & \text{ where } (a_1, \ldots, a_n) = (b_1, \ldots, b_n) \Rightarrow t' \text{ in } T \text{ by } \text{treduce 1} \\
& \quad \Rightarrow t[b_1, \ldots, b_n/a_1, \ldots, a_n] = t' \text{ in } T \\
\text{H} \Rightarrow (\lambda x, y, t)(a, b) & \text{ where } t' \text{ in } T \text{ by } \text{treduce 1} \\
& \quad \Rightarrow t[a, b/x, y] = t' \text{ in } T \\
\text{H} \Rightarrow t & \text{ where } (a_1, \ldots, a_n) = (b_1, \ldots, b_n) \text{ by } \text{tcompute} \\
& \quad \Rightarrow t[b_1, \ldots, b_n/a_1, \ldots, a_n] \\
\text{H} \Rightarrow (\lambda x, y, t)(a, b) & \text{ by } \text{tcompute} \\
& \quad \Rightarrow t[a, b/x, y] \\
\text{H} \Rightarrow t & \text{ where } (a_1, \ldots, a_n) = (b_1, \ldots, b_n) \Rightarrow t' \text{ in } T, H' \Rightarrow T' \\
& \quad \Rightarrow H, t[b_1, \ldots, b_n/a_1, \ldots, a_n] = t' \text{ in } T, H' \Rightarrow T' \\
\text{H} \Rightarrow t & \text{ where } (a_1, \ldots, a_n) = (b_1, \ldots, b_n) \Rightarrow t' \text{ in } T, H' \Rightarrow T' \\
& \quad \Rightarrow H, t[a, b/x, y], H' \Rightarrow T' \\
\end{align*}} \]

\[ \text{II. Tactics} \]
\[ \text{\begin{align*}
\text{COMPUTATION} \\
\text{H} \Rightarrow f(x, \text{redex}) & \Rightarrow t \text{ in } T \\
& \quad \Rightarrow f(x, \text{contractum}) \Rightarrow t \text{ in } T \\
\text{H} \Rightarrow t & \Rightarrow f(x, \text{redex}) \text{ in } T \\
& \quad \Rightarrow t = f(x, \text{contractum}) \text{ in } T \\
\text{H} \Rightarrow f(\text{redex}, w) & \Rightarrow t \text{ in } T \\
& \quad \Rightarrow f(\text{contractum}, w) \Rightarrow t \text{ in } T \\
\text{H} \Rightarrow t & \Rightarrow \text{f(\text{redex}, w}) \Rightarrow t \text{ in } T \\
& \quad \Rightarrow t = f(\text{\text{contractum}, w}) \text{ in } T \\
\text{H} \Rightarrow \text{\text{redex}(x, w)} & \Rightarrow t \text{ in } T \\
& \quad \Rightarrow \text{contractum}(x, w) \Rightarrow t \text{ in } T \\
\text{H} \Rightarrow t & \Rightarrow \text{\text{\text{redex}(x, w}) \Rightarrow t \text{ in } T \\
& \quad \Rightarrow t = \text{\text{\text{contractum}(x, w}) \Rightarrow t \text{ in } T \
\end{align*}} \]
Term constructors, predicates and destructors

---

a: Constructors
---

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>make_SETS_term/SETS</td>
<td>term</td>
<td>&quot;SETS&quot;</td>
</tr>
<tr>
<td>make_carrier_term</td>
<td>term -&gt; term</td>
<td>S into &quot;carrier(S)&quot;</td>
</tr>
<tr>
<td>make_pred_term</td>
<td>term -&gt; term</td>
<td>S into &quot;pred(S)&quot;</td>
</tr>
<tr>
<td>make_setdef_term</td>
<td>term -&gt; term</td>
<td>x A B into &quot;{x:A}</td>
</tr>
<tr>
<td>make_singleton_term</td>
<td>term -&gt; term</td>
<td>x A into &quot;{x:A}&quot;</td>
</tr>
<tr>
<td>make_inset_term</td>
<td>term -&gt; term</td>
<td>x S into &quot;x e S&quot;</td>
</tr>
<tr>
<td>make_set_conversion</td>
<td>term -&gt; term</td>
<td>S into &quot;{S}&quot;</td>
</tr>
<tr>
<td>make_powerset_term</td>
<td>term -&gt; term</td>
<td>A into &quot;P(A)&quot;</td>
</tr>
</tbody>
</table>

b: Selection and predicates as usual

I. RULES
---

help tactic
---

SETS_spread = tactic

Ia: Formation
---

H >> SETS in U1 by Sintro (\{\}) SETSEquality / SetSeq

H >> \{S\} in U1 by Sintro
   >> S in SETS

H >> P(A) in U1 by Sintro (\{\}) SETSEquality_power / Spower
   >> A in TYPE

H >> x e \{S\} in U1 by Sintro
   >> S in SETS
   >> x in carrier(S)

Ib: Introduction
---

H >> (x,P) in SETS by Sintro
   >> X in TYPE
   >> P;x -> U1

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

H >> x in \{S\} by Sintro
   >> x in carrier(S)
   >> pred(S)(x)

H >> carrier(S) in U1 by Sintro
   >> S in SETS

H >> pred(S) in carrier(S) -> U1 by Sintro
   >> S in SETS
Ic: Elimination
-------------
\[
H, S : \text{SETS}, H' \gg T \\
\text{carrier: \text{TYPE}, pred:carrier} \rightarrow \text{U1, } S = (\text{carrier}, \text{pred}) \text{ in SETS} \\
\gg T[(\text{carrier}, \text{pred})/S]
\]
\[
H, x : S, H' \gg T \\
x \text{ in carrier}(S), x \in S \gg T
\]
conversion_elim hyp
\[
H, S : P(A), H' \gg T \\
A \text{ in \text{TYPE}} \\
pred:A \rightarrow \text{U1, } S = (A, \text{pred}) \text{ in SETS} \gg T[(A, \text{pred})/S]
\]
powerset_elim hyp
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
Constructors, Destructors & Predicates

a: Constructors

---

\begin{verbatim}
make_leq_term : term -> term -> term  \rightarrow \text{i j into "\{i\leq\ j\}"}
make_uneq_term : term -> term -> term  \rightarrow \text{i j into "\{i\neq\ j\}"}
make_nat_term/nat : term -> term -> term  \rightarrow \text{\"N\"}
make_nbar_term : term -> term -> term  \rightarrow \text{\"\{0,...,j\}\"}
make_finite_term/finite_sets : term -> term  \rightarrow \text{\"FINITE SETS\"}
make_finset_term : term -> term  \rightarrow \text{\"\{A\}\"}
\end{verbatim}

Destructors and predicates as usual

Ia: Formation

---

\begin{verbatim}
H \gg \text{i j in U i by fintro}
   \gg i in int
   \gg j in int
\end{verbatim}

\begin{verbatim}
leq_intro
\end{verbatim}

\begin{verbatim}
H \gg \text{i j in U i by fintro}
   \gg i in int
   \gg j in int
\end{verbatim}

\begin{verbatim}
uneq_intro
\end{verbatim}

\begin{verbatim}
H \gg \text{N in U i by fintro}
   \gg n in int
\end{verbatim}

\begin{verbatim}
nat_intro
\end{verbatim}

\begin{verbatim}
H \gg \text{(1,...,n) in U i by fintro}
   \gg n in int
\end{verbatim}

\begin{verbatim}
a_bar_intro
\end{verbatim}

\begin{verbatim}
H \gg \text{(0,...,n) in U i by fintro}
   \gg n in int
\end{verbatim}

\begin{verbatim}
bar1_intro
\end{verbatim}

\begin{verbatim}
H \gg \text{FINITE SETS in U i by fintro}
\end{verbatim}

\begin{verbatim}
finset_intro
\end{verbatim}

\begin{verbatim}
H \gg \text{(A) in U i by fintro}
   \gg A in FINITE SETS
\end{verbatim}

\begin{verbatim}
finset_equality_conversion
\end{verbatim}

Ib: Introduction

---

Ic: Elimination

---

Id: Computation

---

these should include rules like (see theorems in chapter 12b/12c)

\begin{verbatim}
H \gg \text{x in \{1,...,k+1\}}
   \gg x \in \{1,...,k\}
\end{verbatim}

\begin{verbatim}
H \gg \text{x in \{1,...,(k+1)-1\}}
   \gg x \in \{1,...,k\}
\end{verbatim}

\begin{verbatim}
H \gg \text{x in \{1,...,k-1\}}
   \gg x \in \{1,...,k\}
   \gg x \leq k
\end{verbatim}

\begin{verbatim}
H \gg \text{0 < j-1}
   \gg i \leq j
\end{verbatim}
### 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_steps_term` = term -> term : w into 'w#e'  
- `make_eps_term/eps` = term : 'e'  
- `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[1+i..lg]'  
- `make_select_term` = term -> term -> term : w i into 'w(i)'  
- `make_cutsuffix_term` = term -> term -> term : w i into 'w[i..]'  
- `make_range_term` = term -> term -> term -> term : w i j into 'w[i+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.

- `destructor_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`)  
- `select`  
- `range`  
- `cutsuffix`

me has to remember this if one does case analysis using word-terms.
I. RULES

Ia: Formation

\[ H \Rightarrow \text{SYMBOLS in } U \]
by \text{wintro} \quad \text{symbol\_equality} / \text{sequal}

\[ H \Rightarrow \text{WORDS in } U \]
by \text{wintro} \quad \text{word\_equality} / \text{wequal}

\[ H \Rightarrow w \in \text{WORDS} \]
by \text{wintro} \quad \text{noteps\_equality} / \text{noteps}

Ib: Introduction

\[ H \Rightarrow e \in \text{WORDS} \]
by \text{wintro} \quad \text{word\_equality\_eps} / \text{weps}

\[ H \Rightarrow a.v \in \text{WORDS} \]
\[ \Rightarrow a \in \text{SYMBOLS} \]
\[ \Rightarrow v \in \text{WORDS} \]
by \text{wintro} \quad \text{word\_equality\_cons} / \text{wcons}

\[ H \Rightarrow u.v \in \text{WORDS} \]
\[ \Rightarrow u \in \text{WORDS} \]
\[ \Rightarrow v \in \text{WORDS} \]
by \text{wintro} \quad \text{word\_equality\_concat} / \text{wconcat}

\[ H \Rightarrow a.a \in \text{WORDS} \]
\[ \Rightarrow a \in \text{SYMBOLS} \]
by \text{wintro} \quad \text{word\_equality\_symbols} / \text{wSYM}

\[ H \Rightarrow u.a \in \text{WORDS} \]
\[ \Rightarrow u \in \text{WORDS} \]
\[ \Rightarrow a \in \text{SYMBOLS} \]
by \text{wintro} \quad \text{word\_equality\_anticons} / \text{wanTI}

\[ H \Rightarrow w \in \text{WORDS} \]
\[ \Rightarrow w \in \text{WORDS} \]
by \text{wintro} \quad \text{word\_equality\_rev} / \text{wrev}

\[ H \Rightarrow w.1 \in \text{WORDS} \]
\[ \Rightarrow 1 \in \text{int} \]
\[ \Rightarrow w \in \text{WORDS} \]
by \text{wintro} \quad \text{word\_equality\_iter} / \text{witer}

\[ H \Rightarrow h.d(w) \in \text{SYMBOLS} \]
\[ \Rightarrow w \in \text{WORDS} \]
by \text{wintro} \quad \text{word\_equality\_hd} / \text{whd}

\[ H \Rightarrow t.l(w) \in \text{WORDS} \]
\[ \Rightarrow w \in \text{WORDS} \]
by \text{wintro} \quad \text{word\_equality\_tl} / \text{wTL}

\[ H \Rightarrow [w] \in \text{int} \]
\[ \Rightarrow w \in \text{WORDS} \]
by \text{wintro} \quad \text{word\_equality\_lg} / \text{wLG}

\[ H \Rightarrow w[i...lg] \in \text{WORDS} \]
\[ \Rightarrow j \in \text{int} \]
\[ \Rightarrow w \in \text{WORDS} \]
by \text{wintro} \quad \text{word\_equality\_cutprefix} / \text{wpre}

\[ H \Rightarrow w(i) \in \text{SYMBOLS} \]
\[ \Rightarrow i \in \text{int} \]
\[ \Rightarrow w \in \text{WORDS} \]
by \text{wintro} \quad \text{word\_equality\_select} / \text{wselect}

\[ H \Rightarrow w[i...1] \in \text{WORDS} \]
\[ \Rightarrow i \in \text{int} \]
\[ \Rightarrow w \in \text{WORDS} \]
by \text{wintro} \quad \text{word\_equality\_cutsuffix} / \text{wSuf}

\[ H \Rightarrow w[i...r] \in \text{WORDS} \]
\[ \Rightarrow i \in \text{int} \]
\[ \Rightarrow r \in \text{int} \]
\[ \Rightarrow w \in \text{WORDS} \]
by \text{wintro} \quad \text{word\_equality\_range} / \text{wrange}
Ic: Elimination

---------

There is no SYMBOL elimination

\[ H, w : \text{WORDS}, H^* \Rightarrow T \]
\[ \Rightarrow T[e/w] \]
\[ \text{hd:SYMBOLS, tl:WORDS, t1_hyp:T}[t1/w] \Rightarrow T[hd.tl/w] \]

\[ H, w : \text{WORDS}, H^* \Rightarrow T \]
\[ u : \text{WORDS} \Rightarrow T[u/w] \text{ in U10} \]
\[ \Rightarrow T[e/w] \]
\[ a : \text{SYMBOLS, v:WORDS, T[v/w]} \Rightarrow T[v\cdot a/w] \]

\[ H, w : \text{WORDS}, H^* \Rightarrow T \]
\[ u : \text{WORDS} \Rightarrow T[u/w] \text{ in U10} \]
\[ \Rightarrow T[e/w] \]
\[ i : \text{int}, 0<i, (\forall v: \text{WORDS}. |v|=i-1 \text{ in int } \Rightarrow T[v/w]), u : \text{WORDS}, |u|=1 \text{ in int} \]
\[ \Rightarrow T[u/w] \]

Additional Helpfunctions
-----------------------------
\text{word}\_\text{equality}\_\text{induction} : \text{tactic}
\text{word}\_\text{integer}\_\text{induction} : \text{tactic}
I. RULES

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

Id: Computation

H  >> e = w in WORDS by wreduce 1
    >> e = w in WORDS
H  >> v = w in WORDS by wreduce 1
    >> v = w in WORDS
H  >> v = w in WORDS by wreduce 1
    >> v = w in WORDS
H  >> a.v = w in WORDS by wreduce 1
    >> a.v = w in WORDS
H  >> a.(u.v) = w in WORDS by wreduce 1
    >> a.(u.v) = w in WORDS
H  >> u.v = w in WORDS by wreduce 1
    >> u.v = w in WORDS
H  >> u = w in WORDS by wreduce 1
    >> u = w in WORDS
H  >> a = w in WORDS by wreduce 1
    >> a = w in WORDS
H  >> a = w in WORDS by wreduce 1
    >> a = w in WORDS
H  >> a = w in WORDS by wreduce 1
    >> a = w in WORDS
H  >> u = w in WORDS by wreduce 1
    >> u = w in WORDS
H  >> a = w in SYMBOLS by wreduce 1
    >> a = w in SYMBOLS
H  >> a = w in SYMBOLS by wreduce 1
    >> a = w in SYMBOLS
H  >> e = x in int by wreduce1
    >> e = x in int
H  >> a = x in int by wreduce1
    >> a = x in int
H  >> a = x in int by wreduce1
    >> a = x in int
H \triangleright |v| \equiv x \text{ in } \text{int} \text{ by wreduce 1} \quad \text{lg_concat no}

H \triangleright |u\cdot a| \equiv x \text{ in } \text{int} \text{ by wreduce 1} \quad \text{lg_anticons no}

H \triangleright |v| \equiv x \text{ in } \text{int} \text{ by wreduce 1} \quad \text{lg_rev no}

H \triangleright |v[1..lg]| \equiv x \text{ in } \text{int} \text{ by wreduce 1} \quad \text{lg_pre no}

H \triangleright |v[1..r]| \equiv x \text{ in } \text{int} \text{ by wreduce 1} \quad \text{lg_suf no}

H \triangleright |v[1+1..r]| \equiv x \text{ in } \text{int} \text{ by wreduce 1} \quad \text{lg_range no}

H \triangleright \text{v} \cdot \text{w} \equiv \text{w} \text{ in } \text{WORDS by }
\quad \varepsilon = \text{w} \text{ in } \text{WORDS} \\
\quad m < 0 
\quad \text{iter_down no} \quad \text{(NOT INCLUDED in wreduce !)}

H \triangleright \text{v} \cdot \text{0} \equiv \text{w} \text{ in } \text{WORDS by wreduce 1} 
\quad \varepsilon = \text{w} \text{ in } \text{WORDS} 
\quad \text{iter_base no}

H \triangleright \text{v} \cdot \text{1} \equiv \text{w} \text{ in } \text{WORDS by wreduce 1} 
\quad \varepsilon = \text{w} \text{ in } \text{WORDS} 
\quad \text{iter_up no}

H \triangleright \text{v} \cdot \text{1} \equiv \text{w} \text{ in } \text{WORDS by wreduce 1} 
\quad \varepsilon = \text{w} \text{ in } \text{WORDS} 
\quad \text{iter_1 no}

H \triangleright \text{w}[0+1..lg] \equiv \text{w} \text{ in } \text{WORDS by wreduce 1} 
\quad \text{v} = \text{w} \text{ in } \text{WORDS} 
\quad \text{pre_base no}

H \triangleright \text{w}[k+1..lg] \equiv \text{w} \text{ in } \text{WORDS by wreduce 1} 
\quad \text{t}(\text{v}[(k-1)+1..lg]) = \text{w} \text{ in } \text{WORDS} 
\quad 0 < k 
\quad \text{pre_up no}
\quad \text{pre_down no} \quad \text{(NOT INCLUDED in wreduce !)}

H \triangleright \text{t}(\text{v}[k+1..lg]) \equiv \text{w} \text{ in } \text{WORDS by wreduce 1} 
\quad \text{t}(\text{v}[(k+1)+1..lg]) = \text{w} \text{ in } \text{WORDS} 
\quad k > 0 
\quad \text{pre_tl no}

H \triangleright \text{t}(\text{v}[k+1..lg]) \equiv \text{w} \text{ in } \text{WORDS by wreduce 1} 
\quad \text{t}(\text{v}[(k+1)+1..lg]) = \text{w} \text{ in } \text{WORDS} 
\quad k \text{ in } \text{N} 
\quad \text{pre_top no}
\[
H \gg v(1') = b \text{ in } \text{SYMBOLS by } \text{wreduce 1}
\]
\[
\gg \text{hd}(v) = b \text{ in } \text{SYMBOLS}
\]
\[
\gg v \text{ in } \text{WORDS}
\]

\[
H \gg (a \cdot v)(i+1) = b \text{ in } \text{SYMBOLS by } \text{wreduce 1}
\]
\[
\gg v(1) = b \text{ in } \text{SYMBOLS}
\]
\[
\gg a \text{ in } \text{SYMBOLS}
\]
\[
0 < i+1
\]

\[
H \gg v[1..m] = w \text{ in } \text{WORDS by }
\]
\[
\gg c = \text{w in } \text{WORDS}
\]
\[
\gg m < 0
\]

\[
H \gg v[1..0] = w \text{ in } \text{WORDS by } \text{wreduce 1}
\]
\[
\gg c = \text{w in } \text{WORDS}
\]

\[
H \gg v[1..r] = w \text{ in } \text{WORDS by } \text{wreduce 1}
\]
\[
\gg (v[1..r-1] \cdot v(r)) = w \text{ in } \text{WORDS}
\]
\[
0 < r
\]

\[
H \gg v[1..|v|] = w \text{ in } \text{WORDS by } \text{wreduce 1}
\]
\[
\gg v = w \text{ in } \text{WORDS}
\]

\[
H \gg v[0+1..r] = w \text{ in } \text{WORDS by } \text{wreduce 1}
\]
\[
\gg v[1..r] = w \text{ in } \text{WORDS}
\]

\[
H \gg v[1..r] = w \text{ in } \text{WORDS by }
\]
\[
\gg v[0+1..r] = w \text{ in } \text{WORDS}
\]

\[
H \gg v[1+1..|v|] = w \text{ in } \text{WORDS by } \text{wreduce 1}
\]
\[
\gg v[1+1..|v|] = w \text{ in } \text{WORDS}
\]

\[
H \gg v[1+1..l] = w \text{ in } \text{WORDS by }
\]
\[
\gg v[1+1..|v|] = w \text{ in } \text{WORDS}
\]

\[
H \gg v(1) \cdot v[1+1..r] = w \text{ in } \text{WORDS by }
\]
\[
\gg v[1+1..r] = w \text{ in } \text{WORDS}
\]

\[
H \gg v[1+1..r] = w \text{ in } \text{WORDS by } \text{wreduce 1}
\]
\[
\gg v(1) \cdot v[1+1..r] = w \text{ in } \text{WORDS}
\]
\[
\gg v[1+1..r+1] = w \text{ in } \text{WORDS}
\]

\[
H \gg v[1+1..r+1] = w \text{ in } \text{WORDS by } \text{wreduce 1}
\]
\[
\gg v[1+1..r+1] = w \text{ in } \text{WORDS}
\]

\[
H \gg v[r+1..r] = w \text{ in } \text{WORDS by } \text{wreduce 1}
\]
\[
\gg c = \text{w in } \text{WORDS}
\]
\[
\gg v \text{ in } \text{WORDS}
\]

\[
H \gg v[0+1..|v|] = w \text{ in } \text{WORDS by } \text{wreduce 1}
\]
\[
\gg v = w \text{ in } \text{WORDS}
\]

\[
H \gg v[1+1..k] = v[k+1..r] = w \text{ in } \text{WORDS by } \text{wreduce 1}
\]
\[
\gg v[1+1..r] = w \text{ in } \text{WORDS}
\]
\[
\gg 1 \text{ in } \text{int}
\]
\[
\gg k \text{ in } \text{int}
\]
\[
\gg r \text{ in } \text{int}
\]
\[
\gg 0 \leq l
\]
\[
\gg l \leq k
\]
\[
\gg k \leq r
\]
CONVERSIONS (to be written)
-----------

: H >> 3z:WORDS, 3b:S SYMBOLS. z\#b = w in WORDS
   >> w#c

: H >> w#c
   >> 3z:WORDS, 3b:S SYMBOLS. z\#b = w in WORDS

: H >> | ψ = 0 'in' int
   >> w = c in W#B

: H >> w = c 'in' WORDS
   >> | ψ = 0 'in' int

: H >> 0 < | ψ
   >> w#c

: H >> w#c
   >> 0 < | ψ

: H >> w#c
   >> ~( w = c 'in' W#RS)

: H >> ( w = c 'in' WORDS)
   >> w#c

Help function
 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
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 >> va:S SYMBOLS.T
   a:S SYMBOLS >> T symbol_all_intro

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

H >> vw:WORDS.T
   >> T[v/w]

H >> vw:WORDS.T
   u:WORDS >> T[u/w] in U10
   >> T[v/w]

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

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

H >> vw:WORDS.T
   >> T[v/w]
   / tail_induction

H >> vw:WORDS.T
   >> T[v/w]
   i:int, 0<i, (Vv:WORDS. |v|=i-1 in int => T[v/w]), u:WORDS, |u|=i in int
   >> T[u/w] lg_induction_using_tac
   / lg_induction

IIc: Some introduction for WORDS

H >> w1,...,wn:WORDS.T
   w1,...,wn:WORDS >> T in U1
   >> T[v1,...,vn/w1,...,wn] word_some_intro level [v1;...;vn]
   ("vi in WORDS" must be provable by Wmember)

IId: Not eps Introduction

H >> a:1
   >> a in SYMBOLS
   >> 1 in WORDS word_intro_noteps / wnoteps

IIe: a special variant of con_asoz

H >> (u+v)+w = x(y+z) in WORDS by con_asoz 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 \gg TRK(g,h) in (A#WORDS) -> B
   \gg A in TYPE
   \gg B in TYPE
   \gg g in A -> B
   \gg h in (B#SYMBOLS) -> B

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

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

Id: Computation

---

H \gg TRK(g,h)(x,e) = t in B by trk_reduce 1
   \gg g(x) = t in B

H \gg TRK(g,h)(x,v*#a) = t in B by trk_reduce 1
   \gg h(TRK(g,h)(x,v),a) = t in B

H \gg h*(x,e) = t in B by trk_reduce 1
   \gg x = t in B

H \gg h*(x,v#a) = t in B by trk_reduce 1
   \gg h(h*(x,v),a) = t in B
   \gg v in WORDS
   \gg a in SYMBOLS
   \gg B in TYPE
   \gg x in B
   \gg h in (B#SYMBOLS) -> B

H \gg h*(x,u#v) = t in B by trk_reduce 1
   \gg h*(h*(x,u),v) = t in B
   \gg v in WORDS
   \gg w in WORDS
   \gg B in TYPE
   \gg x in B

I 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
. Constructors, destructors & predicates

a: Constructors

\[
\begin{array}{rcl}
\text{make_states_term/states} & \rightarrow & \text{"STATES"} \\
\text{make_dfa_term/dfa} & \rightarrow & \text{"DFA"} \\
\text{make-aut_term} & \rightarrow & \text{term} \rightarrow \text{term} \rightarrow \text{term} \rightarrow \text{term} \rightarrow \text{term} \\
\text{make-star_term} & \rightarrow & \text{term} \rightarrow \text{term} \\
\text{make-accept_term} & \rightarrow & \text{term} \rightarrow \text{term} \\
\text{make-accepted_term} & \rightarrow & \text{term} \rightarrow \text{term} \\
\end{array}
\]

Q \rightarrow \text{"d"} \\
M \rightarrow \text{"L(M)"} \\
W \rightarrow \text{"w \in \text{L(M)}"}

Additional functions

\[
\begin{array}{rcl}
\text{make-table_type_term} & \rightarrow & \text{term} \\
\end{array}
\]

Q \rightarrow \text{"((Q)\#\text{SYMBOLS}) \rightarrow \text{(Q)}"}

Destructors & Predicates as usual

I. RULES

---

Helpfunction: 
H, Q:STATES, H' \rightarrow \text{"((Q)\#\text{SYMBOLS}) \rightarrow \text{Q}"} \# Q \# P(Q) \text{ in U} \text{ by dfa_parts_intro (i\#1) }

Ia: Formation

\[
\begin{array}{rcl}
\text{H} & \rightarrow & \text{STATES} \text{ in U} \text{ by dfintr} \text{ o state_intro} \\
\text{H} & \rightarrow & \text{Q} \text{ in U} \text{ by dfintr} \\
\text{H} & \rightarrow & \text{DFA} \text{ in U} \text{ by dfintr} \text{ (i\#1)} \text{ dfa_intro} \\
\text{H} & \rightarrow & \text{L(M)} \text{ in U} \text{ by dfintr} \text{ accept_intro} \\
\text{H} & \rightarrow & \text{(Q,d,q0,qf)} \text{ in DFA} \text{ by dfintr} \\
\text{H} & \rightarrow & \text{d*} \text{ in ((Q)\#\text{SYMBOLS}) \rightarrow \text{Q}} \text{ star_intro} \\
\text{H} & \rightarrow & \text{w} \text{ in L(M) by dfintr} \\
\text{H} & \rightarrow & \text{d*(q0,w) = qf in Q} \text{ acceptEquality} \\
\text{H} & \rightarrow & \text{d*(q,w) in T by dfintr} \\
\end{array}
\]

Ib: Introduction

\[
\begin{array}{rcl}
\end{array}
\]
Ic: Elimination
------------------
\[ H \vdash \text{dfelim}\ M \]
\[ Q:\text{STATES}, d:\{(Q)\#\text{SYMBOLS}\} \rightarrow \{Q\}, q_0:\{Q\}, F: P(\{Q\}), M=\{Q,d,q_0,F\} \text{ in DFA} \]
\[ H \vdash T(\{Q,d,q_0,F\}/z) \]

H \gg t \text{ where } (Q,d,q_0,qf) = M - \text{ in } T[M/z] \text{ by dfintro} 
\[ M \text{ in DFA} \]
\[ Q':\text{STATES}, d':\{(Q')\#\text{SYMBOLS}\} \rightarrow \{Q'\}, q_0':\{Q'\}, qf':\{Q'\}, M'=\{Q',d',q_0',qf'\} \text{ in DFA} \]
\[ t[Q',d',q_0',qf'/\{Q,d,q_0,qf\}] \text{ in } T[\{Q',d',q_0',qf'\}/z] \]

H, \[ w:L(M), H' \gg T \text{ by dfelim } w \]
\[ M \text{ in DFA} \]
\[ v:\text{WORDS}, v \in L(M), v = w \text{ in WORDS} \gg T[v/w] \]

Id: Computation
------------------
\[ \text{H} \gg t \text{ where } (Q,d,q_0,qf) = (Q',d',q_0',qf') - \text{ in } T \text{ by dfreduce1} \]
\[ t[Q',d',q_0',qf'/\{Q,d,q_0,qf\}] = t' \text{ in } T \]

H \gg d(q,e) = q' \text{ in } Q \text{ by dfreduce1} 
\[ q' \text{ in } Q \]

H \gg d(q,v@a) = q' \text{ in } Q \text{ by dfreduce1} 
\[ d(d^*(q,v),a) = q' \text{ in } Q \]
\[ v \in \text{WORDS} \]
\[ a \in \text{SYMBOLS} \]
\[ Q \text{ in } \text{STATES} \]
\[ q \text{ in } \{Q\} \]
\[ d \text{ in } \{(Q)\#\text{SYMBOLS}\} \rightarrow \{Q\} \]

H \gg d(q,v@w) = q' \text{ in } Q \text{ by dfreduce1 CONCAT} 
\[ d^*(d^*(q,v),w) = q' \text{ in } Q \]
\[ v \in \text{WORDS} \]
\[ w \in \text{WORDS} \]
\[ Q \text{ in } \text{STATES} \]
\[ q \text{ in } \{Q\} \]
\[ d \text{ in } \{(Q)\#\text{SYMBOLS}\} \rightarrow \{Q\} \]

II Tactics
---------
\[ H \gg t \text{ where } (Q,d,q_0,qf) = (Q',d',q_0',qf') - \text{ by } \]
\[ t[Q',d',q_0',qf'/\{Q,d,q_0,qf\}] \]

H, \[ t \text{ where } (Q,d,q_0,qf) = (Q',d',q_0',qf') - \]
\[ H' \gg T \text{ by } \]
\[ H, \{Q',d',q_0',qf'/\{Q,d,q_0,qf\}, H' \gg T \]