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 \( U_1 \subseteq U_2 \subseteq U_3 \ldots \), 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 \( A \cdot B \) (cartesian products), \( A+B \) (disjoint union) and \( A \rightarrow B \) (the functions from \( A \) to \( B \)). In addition to these constructors there are dependent products \( x:A \Rightarrow B \) (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 \( x:A+B \) 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. \( a \cdot b \) 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 \( a \cdot b \) was especially added for integer arithmetic. Finally recursive types and the type of partial functions were recently implemented.

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

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

\[
H \Rightarrow T \text{ by <rule-name>}
\]

\[
H_1 \Rightarrow T_1
\]

\[
H_k \Rightarrow T_k
\]

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 \((\forall x.b(x))(a)\) into their canonical form \(b(a)\)) and some miscellaneous rules not associated with a particular type (e.g. *arith*, a decision procedure for arithmetic reasoning). See the account given in chapter 8 of [1] for more details.

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

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

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

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

In order to avoid unnecessarily complicated proofs the actual implementation of theorems from a mathematical theory requires a lot of preparations. Many of the types 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:

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

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

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

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

Experience from working with the PRL system showed that theorems should not be proven without an exhaustive use of tactics. Otherwise a proof tree grows incredibly large and none of the original proof ideas remains visible. For example the first version of the proof of the pumping lemma which did not use tactics except for wellformedness goals resulted in a proof tree of depth greater than 30 and branching degree up to 8 (which came from sequencing in new facts). The new version, presented in chapter 4 has only 13 nodes altogether and reflects the original proof much better.
Using tactics and tacticals allows to combine all the formal steps one can overlook at the current node into a single "logical" step. This might result in a seemingly complicated rule but the resulting proof will be much smaller and thus better to understand - if there was a little bit of 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 \texttt{THEN/THENL} then) into a text buffer, e.g. a dummy \texttt{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 \texttt{immediate} (or \texttt{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 \texttt{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 \texttt{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 \( \forall x_1 : A_1, \ldots, x_n : A_n, B \)) or with a membership subgoal of the form \( t \in T \) which was to be proven by a boring series of introduction steps sometimes giving type dependencies or new identifiers. The tactics HYPOTHESIS, THEOREM and membership use strategies which were a result of studying patterns in my own behavior finding the applicable rules in the manual while sitting in front of the screen. They include ML-functions for creating new identifiers (new), matching of the proofs conclusion against some term (match_subterms) and guessing the type dependencies needed in a particular rule (type_of). There are some variants of the tactics THEOREM and HYPOTHESIS in which the user has to give some of the information usually to be found by the search strategies. They are helpful if just some new facts have to be introduced or if the strategies 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

\[
(*\text{<comment>*})
\]
\[
\leftarrow\text{<tactic>}
\]
\[
\&\text{<t:tactic>}
\]
\[
\oplus\text{<t:tactic>}
\]
\[
\text{THEOREM } \text{<name>}
\]

\[
== \quad \quad == \quad \quad == \quad \quad ==
\]
\[
\langle t \rangle \text{ THEN autotactic}
\]
\[
\text{THENMEMBER } \langle t \rangle \text{ membertac}
\]
\[
\langle t \rangle \text{ THEN membertac}
\]
\[
\text{thm } \langle \text{name} \rangle
\]
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** by **autotactic** is supposed to be a fast tactic finishing trivial proofs of subgoals while
**membertac** (currently equal to **membership**) handles more complicated proofs. Both tactics are set
previously but they are subject to changes, i.e. improvements.

### 3.2 Logic (logic.ml)

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

\[ \langle p: \text{prop} \rangle \& \langle q: \text{prop} \rangle = ((\langle p \rangle) \# (\langle q \rangle)) \]

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

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

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

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

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

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

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

\[ \langle p: \text{prop} \rangle \leftrightarrow \langle q: \text{prop} \rangle = \langle p \rangle \rightarrow \langle q \rangle \& \langle q \rangle \rightarrow \langle p \rangle \]

false

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

**TYPE**

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 \( H \Rightarrow 3x1:A1..3xn:An.B \) by giving witnesses \( a_1, \ldots, a_n \) step by step would result in
the following subgoals

\[ \langle a_1 \rangle \text{ in } A_1 \]

\[ \langle a_0 \rangle \text{ in } A_n[a_1, \ldots, a_{n-1}/x_1, \ldots, x_{n-1}] \]

\[ B[a_1, \ldots, a_n/x_1, \ldots, x_n] \rightarrow (3x2:A2..3x_n:An.B) \text{ in } U_{i} \]

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

By sequencing goals like \( \forall x_1:A_1..\forall x_n:An.(B \text{ in } U_{i}) \) 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:

\[
H \Rightarrow \exists x_1 : A_1 \ldots \exists x_n : A_n . B \text{ by } \texttt{repeat\_some\_intro} \ i \ [a_1 : \ldots : a_n] \\
\Rightarrow A_1 \text{ in } U_i \\
x_1 : A_1 \Rightarrow A_2 \text{ in } U_i \\
\ldots \\
x_1 : A_1 , \ldots , x_n : A_n \Rightarrow B \text{ in } U_i \\
\Rightarrow a_1 \text{ in } A_1 \\
\ldots \\
\Rightarrow a_n \text{ in } A_n[a_1 , \ldots , a_{n-1} / x_1 , \ldots , x_{n-1} ] \\
\Rightarrow B[a_1 , \ldots , a_n / x_1 , \ldots , x_n ]
\]

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(al) Allintro, for example, reduces the number of subgoals after introducing a universal quantifier to one.

\[
H \Rightarrow \forall x : A . B \text{ by } \texttt{AllIntro} j \texttt{AUjtec} \\
x : A \Rightarrow B 
\]

provided that AUjtec is able to prove A in Uj completely. Allintro therefore is the key tactic to straightforward introduction of all-quantifiers in theorems. There are variants for particular types (like \texttt{int} or \texttt{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.

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

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 \( y_2 : A_2[A/a/y_1] \# y_3[A/a/y_1] \# A_4[A/a/y_1] \) and \( x_2 = (c, d) \) in \( y_3 : A_3[a,b/y_1,y_2] \# A_4[a,b/y_1,y_2] \) into the one hypothesis one is interested in, which is \( x = (a,b,c,d) \) in \( y_1 : A_1 \# y_2 : A_2 \# y_3 : A_3 \# A_4 \), and computing "using"-types in intermediate steps. Some additional computation rules and tactics are special formulations of the general PRL-rules compute and computehyp.
The extension so far only consists of the definitions and tactics which were necessary for my purposes. It should be completed to a more powerful tool.

### 3.4 Sets

(sets.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 $A \cap B = \{x | x \in A \land x \in B\}$. 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 $\mathbf{x} \mathbf{in} A$ 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 $(\text{carrier},\text{pred})$ where carrier gives the context and pred is a predicate on carrier expressing the relation $x \in A$. In analogy to conventional set-notation the definition $(\{x : \text{carrier} | p\})$ stands for the pair $(\text{carrier}, x, p)$. A conversion from pairs into a set-type of PRL is necessary if access to the elements of a SET is required.

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

```plaintext
SETS

\{<x : \text{id}> : <\text{carrier} : \text{TYPE}> | <\text{pred} : \text{TYPE}>\} == \{\text{carrier} , (\lambda \langle x, \text{pred}\rangle)\}
\{<x, \text{el} : \text{element}> : <\text{carrier} : \text{TYPE}>\} == \{<x, \text{el} : \text{carrier}>, \langle x, \text{el} = \text{el}\rangle \in \text{carrier}\}
\text{carrier}\langle <X : \text{SETS}>\rangle
== \text{carrier} \cdot \text{where} \ (\text{carrier}, \text{pred}) = <X>\cdot
\text{pred}\langle <X : \text{SETS}>\rangle
== \text{pred} \cdot \text{where} \ (\text{carrier}, \text{pred}) = <X>\cdot
\langle x, \text{element} > \in \{<X : \text{SETS}>\}
== \text{pred}\langle <X>\rangle\langle x\rangle
\{<X : \text{SETS} \ \text{into a set}>\}
== \{\langle x : \text{carrier} | \text{pred}(x) \rangle \cdot \text{where} \ (\text{carrier}, \text{pred}) = <X>\cdot
\text{P}\langle <T : \text{TYPE}>\rangle
== \{<S : \text{SETS} | \text{carrier}(S) = <T> \cdot \text{in} \ \text{TYPE}>\}
```

Note that generally the predicate of a SET is not decidable, i.e. $\mathbf{x} \mathbf{e} \mathbf{A} \mathbf{1} - \mathbf{x} \mathbf{e} \mathbf{A}$ does not hold for $\mathbf{x : \text{carrier}}(\mathbf{A})$. If this property is required further conditions have to be given. For example, one could define \textsc{DECIDABLE SETS} $== \{S : \text{SETS} | \ \forall x : \text{carrier}(S).S\}$ and correspondingly decidable powersets as a subset of them. The SET-definitions also could easily be extended to other operations on SETS as follows:

- $\text{AUB} == \{x : \text{carrier}(A) | x \in A \lor x \in B\}$ (Union)
- $\text{A\cap B} == \{x : \text{carrier}(A) | x \in A \land x \in B\}$ (Intersection)
- $\text{AC} == \{x : \text{carrier}(A) | \neg x \in A\}$ (Complement)
- $\text{ACB} == \forall x : \text{carrier}(A).x \in A \rightarrow x \in B$ (Subset)
- $\text{P}(A) == \{S : \text{SETS} | S \subseteq A\}$ (alternate powerset definition)

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

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

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

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

The rules corresponding to these definitions

\[
\begin{align*}
<i: \text{int}> \& <j: \text{int}> & : = (\langle i \rangle - 1) < \langle j \rangle \\
<x1: \text{int}> \& <x2: \text{int}> & : = \neg (\langle x1 \rangle) \& (\langle x2 \rangle) \in \text{int} \\
N & : = \{nn: \text{int} \mid -1 < nn\} \\
\{1, \ldots, \langle \text{bound}: N \rangle\} & : = \{i: \text{int} \mid 0 < i \& i < \langle \text{bound} \rangle\} \\
\{0, \ldots, \langle r: N \rangle\} & : = \{num: \text{int} \mid -1 < num \& num < \langle r \rangle\} \\
\text{FINITE SETS} & : = N \\
\langle A: \text{FINITE SETS into a set} \rangle & : = \{1, \ldots, \langle A \rangle\}
\end{align*}
\]

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

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

* **extend**
  THM \( \forall k: N. \forall y: \{1, \ldots, k\}. \ y \ \text{in} \ \{1, \ldots, k+1\} \)

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

* **restrict1**
  THM \( \forall m: N. \forall x: \{1, \ldots, m\}. \ x \leq m \implies x \ \text{in} \ \{1, \ldots, m-1\} \)

* **finarith**
  THM \( \forall m: N. \forall x: \{1, \ldots, m\}. \ (x = m \ \text{in} \ \{1, \ldots, m\}) \ \text{or} (x < m) \)

* **hole1**
  THM \( \forall k: \text{int}. \forall x: \{1, \ldots, k\}. \forall f: \{1, \ldots, k\} \rightarrow \{1, \ldots, n\}. \forall y: \{1, \ldots, n\} \)
  \( \exists x: \{1, \ldots, k\}. \ f(x) = y \ \text{in} \ \text{int} \ \forall x: \{1, \ldots, k\}. \ f(x) \neq y \)
Some of them deal with problems of integer arithmetic the \texttt{arith}-rule should be able to handle.

* \texttt{int\_arith}
  
  \texttt{THM} $\forall x,y: \text{int}. \ (x = y \ \text{in} \ \text{int}) \ | \ x \neq y$

* \texttt{intsub}
  
  \texttt{THM} $\forall i,j: \text{int}. \ i = j \ \text{in} \ \text{int} \Rightarrow \ j - i = 0 \ \text{in} \ \text{int}$

* \texttt{lessarith}
  
  \texttt{THM} $\forall i,j: \text{int}. \ i < j \Rightarrow 0 < j - i$

* \texttt{lessarithl}
  
  \texttt{THM} $\forall i,j,k: \text{int}. \ i \leq j \ \text{and} \ 0 \leq k \Rightarrow i \leq j + k$

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

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

* \texttt{pigeon}
  
  \texttt{THM} $\forall k: \text{int}. \ 0 < k \Rightarrow \forall f: \{1, \ldots, k+1\} \rightarrow \{1, \ldots, k\}. \ \exists i, j: \{1, \ldots, k+1\}. \ i < j \ \text{and} \ f(i) = f(j) \ \text{in} \ \text{int}$

A proof of this theorem which was built up using only primitive inference rules is sketched in chapter 11 of [1]. The proof presented here demonstrates some of the advantages of exhaustively using tactics over this method. It is much shorter and since parts of it are symmetric but not worth formulating a lemma the method of analogy proofs by just copying and editing text could be applied. It also reflects the original proof much better since PRL-specific technical details like well formedness 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 $0 < k$.

The base case ($k=0$) is easy because there is no function $f: \{1, \ldots, 0+1\} \rightarrow \{1, \ldots, 0\}$.
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. f(\{1,\ldots,k+1\}) \rightarrow \{1,\ldots\})
4. \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 \text{int}
   >> \exists i,j:\{1,\ldots,k+1\}. i<j \& f(i) = f(j) \in \text{int}
BY Cases ['\exists x:\{1,\ldots,k\}. f(x) = f(k+1) \in \text{int}'; '\forall x:\{1,\ldots,k\}. f(x) = f(k+1)']
1* >> (\exists x:\{1,\ldots,k\}. f(x) = f(k+1) \in \text{int}) \mid (\forall x:\{1,\ldots,k\}. f(x) = f(k+1))
2* 5. \exists x:\{1,\ldots,k\}. f(x) = f(k+1) \in \text{int}
   >> \exists i,j:\{1,\ldots,k+1\}. i<j \& f(i) = f(j) \in \text{int}
3* 5. \forall x:\{1,\ldots,k\}. f(x) = f(k+1)
   >> \exists i,j:\{1,\ldots,k+1\}. i<j \& f(i) = f(j) \in \text{int}
```

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

```
EDIT THM pigeon

* top 2 1
1. k:int
2. (\forall k. f(\{1,\ldots,k+1\}) \rightarrow \{1,\ldots\})
4. \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 \text{int}
   >> (\exists x:\{1,\ldots,k\}. f(x) = f(k+1) \in \text{int}) \mid (\forall x:\{1,\ldots,k\}. f(x) = f(k+1))
BY theorem `hole1` ['k':imp;'\forall x.f(x)';'f(k+1)']
THENL [ Elim 5 THENL [\text{Cumulativity};\text{and_elim} 7] ]
   : Elim 5 THENL [\text{Cumulativity};\text{and_elim} 7] ]
   -\text{Normalize \& Hyp 5}
```

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

* pigeon_type
THM >>\forall k:N. \forall f:1{(\ldots,k+1) \rightarrow \{1,\ldots\}}. \forall i,j:\{1,\ldots,k+1\}. (i<j \& f(i) = f(j) \in \text{int}) \text{ in U1}

```
EDIT THM pigeon

* top 2 2
1. k:int
2. (\forall k. f(\{1,\ldots,k+1\}) \rightarrow \{1,\ldots\})
3. f(\{1,\ldots,k+1\}) \rightarrow \{1,\ldots\}
4. \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 \text{int}
5. \exists x:\{1,\ldots,k\}. f(x) = f(k+1) \in \text{int}
   >> \exists i,j:\{1,\ldots,k+1\}. i<j \& f(i) = f(j) \in \text{int}
BY (* choose \( i = x \) from hyp 5 and \( j = k+1 \) *)

some_elim 5
THENL repeat_some_intro 1 ['x':k+1']
THENL [ membership : membership
   : theorem `pigeon_type` ['k':f';'1';j'] ]
   : THEOREM extend : membership
   : -\text{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 2 3 1. In preparation for the next step (top 2 3 2) also the induction hypothesis 4 is unraveled into the hypotheses 5-8.

```
EDIT THM pigeon

* top 2 3
1. k: int
2. 0<k
3. \( f: \{1, \ldots, k+1\} \rightarrow \{1, \ldots, k\} \)
4. \( \forall x: \{1, \ldots, \lfloor k/2 \rfloor \} \rightarrow \{1, \ldots, k-1\} \). \( \exists i: \{1, \ldots, \lfloor k/2 \rfloor \} \)
   \( i < k \) \& \( f(i) = f(j) \) in int
5. \( \forall x: \{1, \ldots, k\}\). \( f(x) = f(k+1) \)
   \( \exists i: \{1, \ldots, k\} \). \( i < k \) \& \( f(i) = f(j) \) in int

BY all_elim_on ['x:int_eq(f(x);k;f(k+1);f(x))'] 4
THENL
  [ Intro THENL [IDTAC:membership ]
   some_elim 5 THEN and_elim 7 THEN NormalizeHyp 8 ]

1* 5. \( x: \{1, \ldots, \lfloor k/2 \rfloor \} \)
   \( \exists i: \{1, \ldots, \lfloor k/2 \rfloor \} \)
   \( i < k \) \& \( f(i) = f(j) \) in int

2* 5. \( i: \{1, \ldots, \lfloor k/2 \rfloor \} \)
   \( j: \{1, \ldots, \lfloor k/2 \rfloor \} \)
   \( i < j \) \& \( f(i) = f(j) \) in int

EDIT THM pigeon

* top 2 3 1
1. k: int
2. 0<k
3. \( f: \{1, \ldots, k+1\} \rightarrow \{1, \ldots, k\} \)
4. \( \forall x: \{1, \ldots, k\} \). \( f(x) = f(k+1) \)
5. \( \forall x: \{1, \ldots, \lfloor k/2 \rfloor \} \)
   \( \exists i: \{1, \ldots, \lfloor k/2 \rfloor \} \)
   \( i < k \) \& \( f(i) = f(j) \) in int

BY Seq ['f(x) in \{1, \ldots, k\}']
THENL
  [ theorem 'extend1' ['k';'x']
   : Hyp_elim_on ['x'] ] 4
  THENL
   [ THEOREM fineq
    : refine (int_eq_equality)
    THENL
     [ theorem 'wt1' ['k';'f(x)']
      : Equal
      : THEOREM 'restrict1'
      THENL
       [ membership:membership:theorem 'finfuntype' ['k';'f';'k+1']
         : THEOREM restrict1
       ] ] ]
```
Now again a case analysis has to be done:

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

The validity of theses cases again is mainly a problem of finite-set-arithmetic. Trivial facts like \( f(i) \in \text{int} \) 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 $\text{int_eq}$-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} \rightarrow {1,...,k}
4. Vx:{1,...,k}. f(x)=f(k+1)
5. i:{1,...,(k-1)+1}
6. j:{1,...,(k-1)+1}
7. i<j
8. int_eq(f(i);k;f(k+1);f(i)) = int_eq(f(j);k;f(k+1);f(j)) in int
9. Vx:{1,...,k+1}. f(x) in int
10. f(i) in int
11. f(i)=k
12. f(j) = k in int
>> 3:i,j:{1,...,k+1}. i<j & f(i) = f(j) in int

BY (* choose i=1 and j=k+1 *)
repeat_some_intro 1 ['i':';'k+1']

THENL
[ membership; membership
  : theorem 'pigeon_type ['k':';'f':';'i':';'j']
  : THEOREM extend1; membership
  : -and_intro
THENL
[ Elim 5 THENL [ @Cumulativity; &and_elim 14]

(* i<k+1 *)
  : Seq ['int_eq(f(i);k;f(k+1);f(i)) = f(k+1) in int'
    : 'int_eq(f(j);k;f(k+1);f(j)) = f(i) in int']
  THENL
[ @Hyp elim on ['k+1'] 9 THEN -Inteq_computation 1 true
    ; Equal ] ]

+-----+

EDIT THM pigeon

* top 2 3 2 4
1. k:int
2. 0<k
3. f:{1,...,k+1} \rightarrow {1,...,k}
4. Vx:{1,...,k}. f(x)=f(k+1)
5. i:{1,...,(k-1)+1}
6. j:{1,...,(k-1)+1}
7. i<j
8. int_eq(f(i);k;f(k+1);f(i)) = int_eq(f(j);k;f(k+1);f(j)) in int
9. Vx:{1,...,k+1}. f(x) in int
10. f(i) in int
11. f(i)=k
12. f(j) = k in int
>> 3:i,j:{1,...,k+1}. i<j & f(i) = f(j) in int

BY (* choose i and j as given *)
repeat_some_intro 1 ['i':';'j']

THENL
[ membership; membership
  : theorem 'pigeon_type ['k':';'f':';'i':';'j']
  : THEOREM extend1; THEOREM extend1
  : -and_intro
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(j) in int']
  THENL [-Inteq_computation 1 false; -Inteq_computation 1 false; Equal] ]

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

∀k:int. 0<k → ∀f:{1,...,k+1} → {1,...,k}
. ∃i,j:{1,...,k+1}. i<j & f(i) = f(j) in int

Induction on k.
The down case is contradictory
The base case leads to false

1. Base Case:
   There is no function
   from (1,...,1) to (1,...,0)

2. Consider two cases
   1. ∃x:{1,...,k}. f(x) = f(k+1)
   2. ∀x:{1,...,k}. f(x) ≠ f(k+1)

3. Apply Lemma hole1 to
   show that either one of
   these two cases is true
   1. Choose i = x
      and j = k + 1

4. 2. Use induction hypothesis to show
     int_eq(f(i;k;fk+1);f(i)) =
     int_eq(f(j;k;fk+1);f(j))

3. Show that the function
   ∀x:int_eq(f(x;k;fk+1);f(x))
   is of the required type
   1. f(i) = k & f(j) = k
   2. f(i) = k & f(j) ≠ k
   3. f(i) ≠ k & f(j) = k
   4. f(i) ≠ k & f(j) ≠ k

4. Consider four cases
   1. f(i) = k & f(j) = k
   2. f(i) = k & f(j) ≠ k
   3. f(i) ≠ k & f(j) = k
   4. f(i) ≠ k & f(j) ≠ k

4. 1. Choose i and j
     as given
   2. Choose i = j
      and j = k + 1
   3. Choose i = i
      and j = k + 1
   4. Choose i and j
      as given
3.6 Strings

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

Most of the definitions given below are self-explanatory. `ε` is the empty word; `a` is the word solely consisting of the symbol `a`; `uv` is the concatenation of `u` and `v`; `w[a]` means inserting the symbol `a` at the end of the word `w` (cons at the tail of `w`); `w[::-]` is the word `w` in reverse order; `w[i]` is the `i`-fold iteration of `w`; `hd(w)` is the first symbol of `w`; `tl(w)` is the rest; `|w|` is the length of `w`; `w[i1..lg]` is the substring of `w` resulting from cutting off the first `i1` symbols; `w[i]` is the `i`-th symbol in `w`; `w[1..i]` is the substring of `w` consisting of the first `i` symbols; `w[1+1..r]` is the word `w(1+1) ... w(r)`.

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

ε == nil
(a:SYMBOLS) => WORDS == 3a:SYMBOLS. 31:WORDS. (word) == a.1 in WORDS
(s:SYMBOLS) => (rw:WORDS) => list_ind(1w:rw; hd1.tll.tll_rw.(hd1.tll_rw))
(word:SYMBOLS) => list_ind(1word;c; hw; t.o; 1r.word; t.1r.word; hw)
(word:WORDS) => ind(1t; k1.indhyp. c; k1.indhyp. (word) = indhyp)
hd(word:WORDS) => list_ind(1word; 0; hw; t.o; hw; t.1r.word; ind)
tl(word:WORDS) => list_ind(1word; c; hw; t.o; hw; t.1r.word; ind)
[word:WORDS] == ind(1t; k1.w; w.l.g; w.l.g; word; k1.w; w.l.g; tl(w.l.g))
(word:SYMBOLS) => [1..(ii:N)] => ind(ii; k1.w.k1_1.c; k1.w.k1_1.c; k1.w.k1_1.c; k1.w.k1_1.c; k1.w.k1_1.c = word(k1))
(word:WORDS) => [1b:N] => +1...[rb:N] = word(1..rb)(b+1..lg)

TRK((g:A->B), h:B#SYMBOLS->B) = ((lx.w.1.list_ind(1w; g((x)): t.l.rev; fw.((h)(fw.endw)))))
[ch1:B#SYMBOLS->B] = TRK(((x.x), (h1))
```

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

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

* This is one of the advantages of defining objects by not using extract terms of theorems

-18-
* trk1
  \[\text{THM} \gg \forall A, B : \text{TYPE. } \forall g : A \to B. \forall h : (B \# \text{SYMBOLS}) \to B. \forall x : A. \text{TRK}(g, h)(x, x) = g(x) \text{ in } B\]

* trk2
  \[\text{THM} \gg \forall A, B : \text{TYPE. } \forall g : A \to B. \forall h : (B \# \text{SYMBOLS}) \to B. \forall x : A. \forall w : \text{WORDS. } \forall a : \text{SYMBOLS}
  \quad \text{TRK}(g, h)(x, x \cdot a) = h(\text{TRK}(g, h)(x, w), a) \text{ in } B\]

* trk3
  \[\text{THM} \gg \forall A, B : \text{TYPE. } \forall g : A \to B. \forall h : (B \# \text{SYMBOLS}) \to B. \forall x : A. \forall w : \text{WORDS. } \text{TRK}(g, h)(x, w) = \text{h}*(g(x), w) \text{ in } B\]

* trk4
  \[\text{THM} \gg \forall B : \text{TYPE. } \forall h : (B \# \text{SYMBOLS}) \to B. \forall u, v : \text{WORDS. } \forall x : B. \text{h}*(x, (u \cdot v)) = \text{h}*(\text{h}*(x, u), v) \text{ in } B\]

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{wintr} which is easier to remember than all the individual rulenames. It is also used to expand the membership tactic to knowledge about \text{WORDS} which results in the very powerful tactic \text{Wmember} where \text{member_tac} uses \text{wintr} 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.

\[
\begin{align*}
\text{H, w:WORDS, H'} & \gg T \\
& \gg T[e/w] \\
& \text{hd:SYMBOLS, t1:WORDS, t1_hyp:T[t1/w]} \gg T[\text{hd}.t1/w] \\
\end{align*}
\]

\[
\begin{align*}
\text{word elim hyp} \\
\end{align*}
\]

\[
\begin{align*}
\text{H, w:WORDS, H'} & \gg T \\
& \gg u:\text{WORDS} \gg T[u/w] \text{ in } \text{U10} \\
& \gg T[e/w] \\
& \text{a:SYMBOLS, v:WORDS, T[v/w]} \gg T[v+a/w] \\
\end{align*}
\]

\[
\begin{align*}
\text{word elim tail hyp} \\
\end{align*}
\]

\[
\begin{align*}
\text{H, w:WORDS, H'} & \gg T \\
& \gg u:\text{WORDS} \gg T[u/w] \text{ in } \text{U10} \\
& \gg T[e/w] \\
& \quad i: \text{int}, 0 < i, (\forall v: \text{WORDS. } |v| = i-1 \text{ in int } \Rightarrow T[v/w]), u: \text{WORDS, } |u| = i \text{ in int} \\
& \gg T[u/w] \\
\end{align*}
\]

\[
\begin{align*}
\text{word elim lg hyp} \\
\end{align*}
\]

The latter rules were written as tactics using the Nuprl-theorems \text{ind_principle} and \text{ind_principle1}.

* \text{ind_principle}
  \[\text{THM} \gg \forall P : \text{WORDS } \to \text{U10. } (P(e) \& \forall i : \text{SYMBOLS. } \forall z : \text{WORDS. } P(z) \Rightarrow P(z \cdot b)) \Rightarrow \forall w : \text{WORDS. } P(w)\]

* \text{ind_principle1}
  \[\text{THM} \gg \forall P : \text{WORDS } \to \text{U10}
  \quad (P(e) \& \forall i : \text{int. } 0 < i \Rightarrow (\forall z : \text{WORDS. } |z| = i-1 \text{ in int } \Rightarrow P(z))) \Rightarrow (\forall z : \text{WORDS. } |z| = i \text{ in int } \Rightarrow P(z))
  \quad \Rightarrow \forall w : \text{WORDS. } P(w)\]

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{U2}. 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.
resulting from it which describe the basic laws of operations on strings. They cover trivial properties like concatenation with the empty word

* \texttt{eps2}

\begin{verbatim}
THM >> \forall v:WORDS. v@e = v \text{ in } WORDS
\end{verbatim}

which yields to the rule

\begin{verbatim}
H >> v@e = w \text{ in words by } \texttt{wreduce 1}
\end{verbatim}

\begin{verbatim}
\texttt{eps_concat_right 1}
\end{verbatim}

or the law of associativity of concatenation

* \texttt{conassoc}

\begin{verbatim}
THM >> \forall u,v,w:WORDS. ((u@v)@w) = (u@(v@w)) \text{ in } WORDS
\end{verbatim}

\begin{verbatim}
H >> (u@v)@z = w \text{ in words by } \texttt{wreduce 1}
\end{verbatim}

\begin{verbatim}
\texttt{con_asoz 1}
\end{verbatim}

the reverse of a concatenated word

* \texttt{revcon}

\begin{verbatim}
THM >> \forall u,v:WORDS. *((u@v)) = (@v@u) \text{ in } WORDS
\end{verbatim}

\begin{verbatim}
H >> *+(u@v) = w \text{ in words by } \texttt{wreduce 1}
\end{verbatim}

\begin{verbatim}
\texttt{rev_con 1}
\end{verbatim}

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

* \texttt{lgprop1}

\begin{verbatim}
THM >> \forall w:WORDS. w@e <\rightarrow \text{0} < |w|
\end{verbatim}

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

* \texttt{rangethm}

\begin{verbatim}
THM >> \forall w:WORDS. \forall l,r:\text{int. } 0 \leq l \& 1 \leq r \& r \leq |w| \Rightarrow w = w[1..l]@w[l+1..r]@w[r+1..lg] \text{ in } WORDS
\end{verbatim}

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.ml}. 

- 20 -
4. Deterministic finite Automata

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

\[
\begin{align*}
\text{STATES} & \quad \text{== FINITE SETS} \\
\text{DFA} & \quad \text{== } (Q:\text{STATES} \# (\{(Q) \# \text{SYMBOLS}\} \Rightarrow \{Q\} \# \{Q\} \# P(\{Q\})) \\
\langle \text{del}:Q\times\text{SYMBOLS}\rangle \times Q & \quad \text{== TRK}(\{(x,x)\langle \text{del} \rangle \}) \\
\langle \text{word}:\text{WORDS} \rangle \in L(\langle \text{AUT}:\text{DFA} \rangle) & \quad \text{== } d^*(q_0, \langle \text{word} \rangle) \in \{F\} \quad \text{where } (Q, d, q_0, F) = \langle \text{AUT} \rangle \\
L(\langle \text{AUT}:\text{DFA} \rangle) & \quad \text{== } \{w:\text{WORDS} \mid w \in L(\langle \text{AUT} \rangle) \}
\end{align*}
\]

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.

\[
\text{pumping}
\]

\[
\begin{align*}
\text{THM} & \quad \Rightarrow \forall M: \text{DFA}. \ \exists n: \text{int}. \ \forall z: \text{WORDS}. \ z \in L(M) & \& n \leq |z| \\
& \quad \Rightarrow \exists u, v, w: \text{WORDS} \\
& \quad . \ z = (u*v)*w \text{ in } \text{WORDS} & \& |u*v| \leq n & \& 0 < |v| & \& \forall k: \text{int}. \ (u*(v^k)) \ast w \in L(M)
\end{align*}
\]

I preferred the formulation \( \forall z: \text{WORDS}. \ z \in L(M) \Rightarrow \) to \( \forall z: L(M) \) \Rightarrow ... because otherwise the property \( d^*(q_0, z) \in \{F\} \quad \text{where } (Q, d, q_0, F) = M \) 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.

\[
\text{pumping type}
\]

\[
\begin{align*}
\text{THM} & \quad \Rightarrow \forall M: \text{DFA}. \ \forall n: \text{int}. \ \forall z: \text{WORDS}. \ \forall u, v, w: \text{WORDS} \\
& \quad . \ (z = (u*v)*w \ \text{in} \ \text{WORDS} & \& |u*v| \leq n & \& 0 < |v| & \& \forall k: \text{int}. \ (u*(v^k)) \ast w \in L(M) ) \ \text{in u1}
\end{align*}
\]

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 1
  1. $M$ : $\texttt{DFA}$
  2. $Q$ : $\texttt{STATES}$
  3. $d : (Q \times \texttt{SYMBOLS}) \to Q$
  4. $q_0 : Q$
  5. $F : P(Q)$
  6. $(Q, d, q_0, F) = M$ in $\texttt{DFA}$

  $\forall n : \texttt{int.} \ \forall z : \texttt{WORDS.} \ z \in L(M) \ & \ n \leq |z|$
  $\Rightarrow 3 u, v, w : \texttt{WORDS}$
  $\Rightarrow ((u + w) \cdot w) \in \texttt{WORDS} \ & \ |(u + v)| \leq n \ & \ 0 < |v| \ & \ \forall k : \texttt{int.} \ ((u + (v \cdot k)) \cdot w) \in L(M)$

BY (* Let $M = (Q, d, q_0, F)$ be a deterministic finite automaton *) $\texttt{dfa_intro_elim}$

*top
  1. $M$ : $\texttt{DFA}$
  2. $Q$ : $\texttt{STATES}$
  3. $d : (Q \times \texttt{SYMBOLS}) \to Q$
  4. $q_0 : Q$
  5. $F : P(Q)$
  6. $(Q, d, q_0, F) = M$ in $\texttt{DFA}$

$\forall n : \texttt{int.} \ \forall z : \texttt{WORDS.} \ z \in L(Q, d, q_0, F) \ & \ n \leq |z|$
$\Rightarrow 3 u, v, w : \texttt{WORDS}$
$\Rightarrow ((u + w) \cdot w) \in \texttt{WORDS} \ & \ |(u + v)| \leq n \ & \ 0 < |v| \ & \ \forall k : \texttt{int.} \ ((u + (v \cdot k)) \cdot w) \in L(Q, d, q_0, F)$

BY (* choose $n$ to be the number of states in $Q$ *)

Some_intro 'Q'

membership (* shows that $Q$ actually is an integer *)

(Intro THENL
  (* show wellformedness of the goal *)
  [ word_equality
    ; Intro THENL
      [ Intro THENL [-accepted_intro : Wmember]
        ; Intro THENL
          [ wequal : Intro THENL [wequal : Intro THENL [wequal : THEOREM pumping_type]]]]
  ])
)

* $\forall z : \texttt{WORDS.} \ z \in L(Q, d, q_0, F) \ & \ (Q) \leq |z|$
$\Rightarrow 3 u, v, w : \texttt{WORDS}$
$\Rightarrow ((u + w) \cdot w) \in \texttt{WORDS} \ & \ |(u + v)| \leq (Q) \ & \ 0 < |v| \ & \ \forall k : \texttt{int.} \ ((u + (v \cdot k)) \cdot w) \in L(Q, d, q_0, F)$
```
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 $z \in L(0, d, q_0, F) \land (Q) = |z|$ 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

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

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

EDIT THM pumping

1. M:DFA
2. Q:STATES
3. d:((Q)\*SYMBOLS)->(Q)
4. q0:Q
5. F: P(Q)
6. (Q,d,q0,F)=M in DFA
7. z:WORDS
8. z \in L(Q,d,q0,F)
9. (Q)::[z]
10. \exists i,j: \text{int}, \ 0\leq i \land i<j \land j \leq l \land d^*(q_0.z[1..l]) = d^*(q_0.(z[1..i]+z[i+1..j])) \text{ in } (Q)

BY (* apply the pigeonhole principle *)

\[-THEOREM1 'pigeon_variant' ['Q': '(\n.d^*(q_0.z[1..n]))'] \]
\THENL
\[ \text{Intro} \text{ THENL} \text{ [delstar_apply; membership; some_elim 10; then_elim 12; then_Hyp_compute_term 12 1; then_Hyp_compute_snd_term 12 1]} \]
\THENL
\[ \text{repeat some intro 1 ['k'; 'l'] \text{ Intro; Intro; IDTAC; Equal; Equal; -repeat_and_intro}} \]
\THENL
\[ \text{Intro} \text{ THENL} \text{ membership; Intro THENL [-state_set_intro; delstar_apply; delstar_apply]} \]
\[ ; \text{ -SubstFor } 'z[1..k]+z[k+1..l] = z[1..l] \text{ in } WORDS' \]
\THENL
\[ \text{THENL} \text{ [THEOREM range9; Intro THENL [-state_set_intro; delstar_apply; delstar_apply]]} \]
In the second subgoal the proof now is ready for the introduction of the desired three substrings. A special variant of the tactic repeat_some_intro, which spared me of a lot of unnecessary wellformedness-subgoals, was used for that. The theorem pumping_type therefore had to be applied only once. $u = z[1..i]$, $v = z[i+1..j]$ and $w = z[j+1..l]$ were chosen. The theorem requires them to fulfill four conditions which result in the four subgoals of this step.

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 2
1. M: DFA
2. Q: STATES
3. d:((Q)#SYMBOLS)->(Q)
4. q0: Q
5. F:P(Q)
6. (Q,d,q0,F)=M in DFA
7. z:WORDS
8. z ε L(Q,d,q0,F)
9. (Q)= z
10. j:int
11. i:int
12. d*(q0,z[1..i]) = d*(q0,(z[1..i]*z[1..j])) in (Q)
13. 0 ≤ j≤ i
14. i< j
15. i=0
>> |z[1..i]*z[1..j]| ≤ (Q)

BY (* |u=v| = j ≤ q * )

@SubstFor '|(z[1..i]*z[1..j])| = i in int'
THENL SubstFor '|(z[1..i]*z[1..j])| = z[1..j] in WORDS'
THENL [ THEOREM range9; THEOREM lngsf THEN &Arith: Wmember]

EDIT THM pumping

* top 1 1 1 2 3
1. M: DFA
2. Q: STATES
3. d:((Q)#SYMBOLS)->(Q)
4. q0: Q
5. F:P(Q)
6. (Q,d,q0,F)=M in DFA
7. z:WORDS
8. z ε L(Q,d,q0,F)
9. (Q)= z
10. i:int
11. j:int
12. d*(q0,z[1..i]) = d*(q0,(z[1..i]*z[1..j])) in (Q)
13. 0 ≤ j≤ i
14. i< j
15. i=0
>> 0< |z[i+1..j]|

BY (* |v| = j-i > 0 *)

SubstFor '|z[i+1..j]|=j-i in int'
THENL [ THEOREM 1grange; THEOREM lessarith: &Cumulativity]
In the fourth subgoal the definition of $z \in L(M)$ is transformed into its original form $d^*(q_0, z) \in \langle F \rangle$ and the same is done to the conclusion after the introduction of $k$.

BY (*Let $k$ be an integer. Go back to the definition of $z \in L(M)$ resp. $u=v \cdot v \in L(M)\ast$)

int_all_intro THEN Compute 3 THEN Hyp_compute 8 3

**Theorem:** $d^*(q_0, z) \in \langle F \rangle$

1. $k$: int
2. $d^*(q_0, (z[1..i] \cdot (z[i+1..j] \cdot k))) \in \langle F \rangle$

BY (*show: $d^*(q_0, z[1..i] \cdot z[1..j] \cdot k)) = d^*(q_0, (z[1..i] \cdot z[1..j] \cdot k)) \in \langle Q \rangle$

which can be reduced to: $d^*(q_0, (z[1..i] \cdot z[1..j] \cdot k)) = d^*(q_0, (z[1..i] \cdot z[1..j] \cdot k))$ (*)

- SubstFor $d^*(q_0, (z[1..i] \cdot z[1..j] \cdot k)) = d^*(q_0, (z[1..i] \cdot z[1..j] \cdot k))$ in $\langle Q \rangle$

THEN

[ SubstFor $d^*(q_0, (z[1..i] \cdot z[1..j] \cdot k)) = d^*(q_0, (z[1..i] \cdot z[1..j] \cdot k))$ in $\langle Q \rangle$

THEN

[ star_concat_compute 1 THEN star_concat_compute 2 THEN delstar_apply


; delstar_apply THEN theorem `rangethm [z':i';j';imp]

; Cumulativity THEN -intro THENL [-state_set_intro; delstar_apply]

; Elim 5 THENL [membership; ~isinset]

]}

1. $d^*(q_0, z[1..i] \cdot (z[i+1..j] \cdot k)) = d^*(q_0, (z[1..i] \cdot z[1..j] \cdot k)) \in \langle Q \rangle$
EDIT THM pumping

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

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

Int elim 9
THENL
[ SubstFor 'z[i+1..j]+k = c in WORDS'
THENL
[ -iter_down 1 THEN weps
 : -star_reduction_eps 1
 : Cumulativity THEN Intro THENL [-state_set_intro;delstar_apply;delstar_apply]
 ]
 : Compute_apply2_arg2 1 THENL -star_reduction_eps 1
 : IDTAC
 ]

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

EDIT THM pumping

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

BY (* Do the final substitutions

\[
\begin{align*}
&d^*(d^*(q_0,u),v^k) = (\text{by definition of } v^k) \\
&d^*(d^*(q_0,u),v^k) = (\text{properties of } \text{delstar}) \\
&d^*(d^*(q_0,u),v^k) = (\text{hypothesis 7}) \\
&d^*(d^*(q_0,u),v^k) = (\text{induction hypothesis}) \\
&d^*(q_0,u) = (v \neq k)
\end{align*}
\]

SubstFor 'z[i+1..j]+m = (z[i+1..j]+z[i+1..j])+(m-1)' in WORDS'
THENL
[ -iter_up 1 THEN Wmember
 : Concat compute 1 THENL [membership; IDTAC]
 : Cumulativity THEN Intro THENL [-state_set_intro;delstar_apply;delstar_apply]
 ]
THENL SubstFor 'd*(d*(q_0,z[i..i]),z[i+1..j]) = d*(q_0,z[i..i])' in (Q)
THENL
[ -THEOREM1 \text{delprop("Q":d:"z[i..i]","z[i+1..j]":q0}') THEN Wintro
 : Equal
 : Base
 : delstar_apply;delstar_apply]
]
Proof-tree of the pumping-lemma

Let $M = (Q, \delta, q_0, F)$ 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 $i < j \leq n$ such that

$$d^*(q_0, z[i..j]) = d^*(q_0, (z[i+1..j]))$$

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

$z = ((uwv)^*w)$

Clear by definition of $u, v,$ and $w$ (Apply "range theorem")

$|uwv| - j \leq Q$

$|v| = j-i > 0$

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

show:

$$d^*(q_0, u(v^k)w) = d^*(q_0, uv^k w) = d^*(q_0, z) \in F$$

which can be reduced to:

$$d^*(d^*(q_0, u), v^k) = d^*(q_0, uv^k)$$

Proof by induction on $k$

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

Do the final substitutions

$$d^*(d^*(q_0, u), v^k) = (\text{by definition of } v^k)$$

$$d^*(d^*(q_0, u), v^k) = (\text{by properties of delstar})$$

$$d^*(d^*(q_0, u), v^k) = (\text{by "loop" hypothesis})$$

$$d^*(q_0, uv^k) = (\text{by induction hypothesis})$$
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.

```plaintext
(* Edit eval pumping demo *)

let M = (4,λa.1;int_eq(a;1;2;int_eq(q;1;2;int_eq(q;2;1;4));
  int_eq(a;2;int_eq(q;1;3;int_eq(q;3;1;4)); 4 )))
  ,1,(4:(1,...,4))
;

(* Automaton for 2(112)* *)

let M1 = (5,λa.1;int_eq(a;1;5;(1+q) mod 6);
  int_eq(a;2;int_eq(q;1;2;int_eq(q;4;2;5)); 5 ))
  ,1,(2:(1,...,5))
;

(* Evaluation term of the pumping lemma *)

let pump = λM,z.( n ,u,v,w) - where (u,v,w,proof)=pump(z)(axiom,axiom)-
  where (n,pump)=term_of(pumping)(M)-
  ;

let test = pump( M , 1.1.2.2.1.1.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.

```
NuPRL 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 , 2.2.2.2.2.2.1.1.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.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:

\[
\text{NFA} \quad \begin{cases} (Q: \text{STATES} \# ( ((Q) \# \text{SYMBOLS}) \rightarrow P(Q)) \# (Q) \# P(Q)) \\
\langle d: Q \times \text{SYMBOLS} \rightarrow P(Q) \rangle \\
\langle w: \text{WORDS} \rangle \in L(\langle A: \text{NFA} \rangle) \\
L(\langle A: \text{NFA} \rangle) \\
\end{cases}
\]

All the rules dealing with non-deterministic finite automata are essentially the same as the ones for \( \text{dfa} \). The just have to be copied and slightly modified. However the Nuprl-proof of the equivalence between \( nfa \) and \( dfa \)

\[\forall M: \text{NFA}. \exists A: \text{DFA}. L(M) = L(A) \text{ in } \text{TYPE} \]

requires some additional definitions which handle the simulation of \( P(Q) \) within the concept of finite sets and an expansion \( d^* \) 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 \( Q \).

\[2^<i: \text{int}>\]
\[\text{no}(Q: \text{STATES}, \langle A: P(Q) \rangle \rightarrow (1 \ldots 2^Q))\]
\[\begin{cases}
\text{set}(Q: \text{STATES}, \langle n: 1 \ldots 2^Q \rangle \rightarrow P(Q))
\end{cases}\]
\[d^*(Q: \text{STATES}, \langle A: P(Q) \rangle, \langle w: \text{WORDS} \rangle)\]

Theorems and tactics have to be written which deal with the most important properties of these definitions, e.g. \( \text{no} \langle \text{set}(Q, n) \rangle = n \text{ in } \langle Q \rangle \). \( \text{set}(Q, \text{no}(A)) = A \text{ in } P(Q) \). \( Q: \text{STATES} \rightarrow 2^Q \text{ in } Q: \text{STATES} \).

\[Q: \text{STATES} \rightarrow d^*(Q, \langle q: Q \rangle, w) = d^*(q, w) \in Q \text{ etc.} \]

After that the proof of the equivalence theorem can easily be written in analogy to the original proof:

1. Let \( M = \langle Q, d, q_0, F \rangle \) be a \( nfa \)
2. Choose \( A = \langle Q_1, d_1, q_1, F_1 \rangle = \langle Q, \lambda q, a. \text{no}(d^*(Q, \langle q, \text{set}(Q, q) \rangle), qa) \rangle, \text{no}(\langle q_0: Q \rangle, \langle f: \{2^Q\} \rangle, \exists g: (\text{set}(Q, f)): g(c(f)) \rangle \rangle \)
3. Show \( \forall w: \text{WORDS}. \exists q: Q. d_1^*(p, w) = \text{no}(d^*(\text{set}(Q, p), w)) \) by tail-induction on \( w \)
4. Reduce \( L(M) = L(A) \text{ in } \text{TYPE} \) using computation rules to \( \exists f: (d^*(q_0, w)). f(c(f)) \quad \text{if} \quad d_1^*(q_1, w) \in \{f: \{2^Q\} \mid \exists g: (\text{set}(Q, f)): g(c(f)) \} \)
5. Proof this by the following substitution sequence:

\[d_1^*(q_1, w) \in \{f: \{2^Q\} \mid \exists g: (\text{set}(Q, f)): g(c(f)) \} \]
\[\rightarrow \text{no}(d^*(Q, \text{set}(Q, \text{no}(\langle q_0: Q \rangle), w))). \text{in} \langle f: \{2^Q\} \mid \exists g: (\text{set}(Q, f)): g(c(f)) \rangle \]
\[\rightarrow \text{no}(d^*(q_0, w)) \in \langle f: \{2^Q\} \mid \exists g: (\text{set}(Q, f)): g(c(f)) \rangle \]
\[\rightarrow \exists g: (\text{set}(Q, \text{no}(d^*(q_0, w))). g(c(f)) \]
\[\rightarrow \exists g: (d^*(q_0, w)). g(c(f)) \]

*It seems that for this conversion it is necessary that \( A \) is a DECIDABLE SET*
The most interesting part of this theorem is that its extract term will actually convert nondeterministic finite automata into the equivalent deterministic ones.

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

\[
\begin{align*}
\varepsilon \cdot cl(<M: \text{NFA}, <n:N>, <q:(Q)>)) & \quad \text{== } \{w:\text{WORDS} | w=\varepsilon \text{ in WORDS}\} \\
\varepsilon \cdot \text{closure}(<M: \text{NFA}, <q:(Q)>)) & \quad \text{== } (Q:\text{STATES} # \{(((Q)\#(\text{SYMBOLS})\#\text{EPS}) \rightarrow P((Q))) \# (Q) \# P((Q))\})
\end{align*}
\]

\[
\begin{align*}
\varepsilon \cdot cl(<M: \text{NFA}, <n:N>, <q:(Q)>)) & \quad \text{== } \text{ind}(<n>: m.\varepsilon \cdot cl(q:(Q))); \ 
\varepsilon \cdot \text{closure}(<M: \text{NFA}, <q:(Q)>)) & \quad \text{== } m.\varepsilon \cdot \text{cl}(q:(Q)); \ 
\varepsilon \cdot \text{closure}(<M: \text{NFA}, <q:(Q)>)) & \quad \text{== } \text{ind}(\varepsilon : r.\varepsilon \cdot \text{cl}(r, \text{inr}(v))); \ 
\varepsilon \cdot \text{closure}(<M: \text{NFA}, <q:(Q)>)) & \quad \text{== } m.\varepsilon \cdot \text{cl}(q:(Q)); \ 
\varepsilon \cdot \text{closure}(<M: \text{NFA}, <q:(Q)>)) & \quad \text{== } \text{ind}(\varepsilon : r.\varepsilon \cdot \text{cl}(r, \text{inr}(v))); \ 
\varepsilon \cdot \text{closure}(<M: \text{NFA}, <q:(Q)>)) & \quad \text{== } m.\varepsilon \cdot \text{cl}(q:(Q));
\end{align*}
\]

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

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

Library Automata

* AUTOMATA THEORY
  DEF PRL-Version by Christoph Kreitz  February 25, 1986

* c1
  DEF

  CHAPTER 1: PRELIMINARIES

* c10
  DEF

  1.0: general definitions for tactics

* c
  DEF (*<comment>*)

* tac
  DEF <tactic>

* tacm
  DEF @<tactic>

* tacmember
  DEF &<tactic>

* theorem
  DEF THEOREM <theorem_name>

* c11
  DEF

  1.1: NOTATION: logic definitions, tupling, functions

* and
  DEF <prop> & <prop>

* all
  DEF \( \forall \var \colon \text{type}. \ <prop> \)

* all2
  DEF \( \forall \var_1, \var_2 \colon \text{type}. \ <prop> \)

* all3
  DEF \( \forall \var_1, \var_2, \var_3 \colon \text{type}. \ <prop> \)

* some
  DEF \( \exists \var \colon \text{type}. \ <prop> \)

* some2
  DEF \( \exists \var_1, \var_2 \colon \text{type}. \ <prop> \)

* some3
  DEF \( \exists \var_1, \var_2, \var_3 \colon \text{type}. \ <prop> \)

* imp
  DEF <prop> \( \rightarrow \) <prop>

* equiv
  DEF <prop> \( \leftrightarrow \) <prop>

* false
  DEF false

* not
  DEF \( \neg \) <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 \

* c12
  DEF
  1.2: SETS

* c12a
  DEF
  1.2.a SETS & POWERSETS

* sets
  DEF SETS

* setdef
  DEF <id>:<TYPE>|<TYPE>|</TYPE>

* singleton
  DEF <element>:<TYPE>

* carrier
  DEF carrier(<SETS>)

* pred
  DEF pred(<SETS>)

* inset
  DEF <element> \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 U_1. S = (A,p) in SETS

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

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

* c12b
  DEF
  1.2.b NATURAL NUMBERS & NUMBER THEORY

* leq
  DEF <int>\leq<int>

* uneq
  DEF <int>\neq<int>

* N
  DEF N

* int_arith
  THM \forall x,y:int. (x = y in int) \iff x=y

* intsub
  THM \forall i,j:int. i=j in int \iff j-i = 0 in int

* lessarith
  THM \forall i,j:int. i<j \iff 0<j-i

* lessarith1
  THM \forall i,j,k:int. i\leq j & 0\leq k \iff i\leq j+k
* **cl2c**
  DEF
  \[ 1.2.c: \text{FINITE SETS} \& \text{Cardinality} \]

* **nbar**
  DEF \( \{1,\ldots,\langle N\rangle\} \)

* **nbar1**
  DEF \( \{0,\ldots,\langle N\rangle\} \)

* **finite**
  DEF FINITE SETS

* **finset**
  DEF \( \langle \text{FINITE SETS into a set} \rangle \)

* **finsettype**
  THM \( \gg \forall A: \text{FINITE SETS}. (A) \text{ in TYPE} \)

* **wf1**
  THM \( \gg \forall n: N. \forall x: \{1,\ldots,n\}. x \text{ in int} \)

* **wf2**
  THM \( \gg \forall n: N. \forall x: \{1,\ldots,n\}. x \leq n \)

* **wf3**
  THM \( \gg \forall n: N. \forall x: \{1,\ldots,n\}. 0 < x \)

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

* **extend**
  THM \( \gg \forall k: N. \forall y: \{1,\ldots,k\}. y \text{ in } \{1,\ldots,k+1\} \)

* **extend1**
  THM \( \gg \forall k: N. \forall x: \{1,\ldots,(k-1)+1\}. x \text{ in } \{1,\ldots,k+1\} \)

* **restrict**
  THM \( \gg \forall m: N. \forall x: \{1,\ldots,m\}. x < m \implies x \text{ in } \{1,\ldots,m-1\} \)

* **restrict1**
  THM \( \gg \forall m: N. \forall x: \{1,\ldots,m\}. x = m \implies x \text{ in } \{1,\ldots,m-1\} \)

* **finarith**
  THM \( \gg \forall m: N. \forall x: \{1,\ldots,m\}. (x = m \text{ in } \{1,\ldots,m\}) \lor (x < m) \)

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

* **hole1**
  THM \( \gg \forall k: \text{int}. 0 < k \implies \forall n: N. \forall f: \{1,\ldots,k\} \rightarrow \{1,\ldots,n\}. \forall y: \{1,\ldots,n\}. \exists x: \{1,\ldots,k\}. f(x) = y \)

* **pigeon type**
  THM \( \gg \forall k: \text{int}. 0 < k \implies \forall f: \{1,\ldots,k+1\} \rightarrow \{1,\ldots,k\}. \forall i, j: \{1,\ldots,k+1\}. (i < j \& f(i) = f(j) \text{ in int}) \in \text{UI} \)

* **pigeon**
  THM \( \gg \forall k: \text{int}. 0 < k \implies \forall f: \{1,\ldots,k+1\} \rightarrow \{1,\ldots,k\}. \exists i, j: \{1,\ldots,k+1\}. (i < j \& f(i) = f(j) \text{ in int}) \)

* **pigeon variant**
  THM \( \gg \forall n: N. \forall h: \{0,\ldots,n\} \rightarrow \{1,\ldots,n\}. \exists k: \text{int}. 0 < k \& k < l \land l \leq n \land h(k) = h(l) \text{ in } \{1,\ldots,n\} \)
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 ⟷ ∀v:WORDS. ∀a:SYMBOLS. (a•v) = a•w in WORDS

symcon
THM ⟷ ∀u,v:WORDS. ∀a:SYMBOLS. (a•u•v) = a•(u•v) in WORDS

conassoz
THM ⟷ ∀u,v,w:WORDS. ((u•v)•w) = (u•(v•w)) in WORDS

epsrev
THM ⟷ ∀ε = ε in WORDS

symrev
THM ⟷ ∀a:SYMBOLS. a•a = a in WORDS
• consrev
  THM \(\forall a:\text{SYMBOLS}. \forall u:\text{WORDS}. \neg a.u = \neg u.a\) in WORDS

• anticonsrev
  THM \(\forall w:\text{WORDS}. \forall a:\text{SYMBOLS}. \neg w.a = a\neg w\) in WORDS

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

• doublerrev
  THM \(\forall u:\text{WORDS}. \neg(\neg u) = u\) in WORDS

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

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

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

• t1
  THM \(\forall a:\text{SYMBOLS}. \forall u,v:\text{WORDS}. \text{tl}((a.u\equiv v)) = (u\equiv v)\) in WORDS

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

• lgtype1
  THM \(\forall w:\text{WORDS}. \text{O}(|w|)\)

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

• lgprop2
  THM \(\forall w:\text{WORDS}. w\epsilon \Rightarrow 0 < |w|\)

• lgeps
  THM \(|\epsilon| = 0\) in int

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

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

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

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

• lgrev
  THM \(\forall w:\text{WORDS}. |\neg w| = |w|\) in int

• lg1
  THM \(\forall w:\text{WORDS}. w\epsilon \Rightarrow |\text{tl}(w)| = (|w|-1)\) in int

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

• lgseq
  THM \(\forall w:\text{WORDS}. \forall i:int. |\epsilon\omega(i)| = 1\) in int

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

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

• iter1
  THM \(\forall w:\text{WORDS}. \forall k:int. k<1 \Rightarrow w\equiv k = \epsilon\) in WORDS

• iter2
  THM \(\forall w:\text{WORDS}. \forall k:int. 0\leq k \Rightarrow w\equiv k = (w\equiv w\equiv k-1)\) in WORDS

• word1
  THM \(\forall v1:\text{WORDS}. \forall a:\text{SYMBOLS}. \exists z:\text{WORDS}. a\equiv b\equiv \text{SYMBOLS}. a.l = z\equiv b\) in WORDS

• notequal1
  THM \(\forall w:\text{WORDS}. (w = \epsilon \text{ in WORDS}) \Rightarrow w\epsilon\)

• notequalsthm
  THM \(\forall w:\text{WORDS}. w\epsilon \Rightarrow \neg w\epsilon \text{ in WORDS}\)
* preeps
  THM >>Vv: int. ε[k+1..lg] = ε in WORDS

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

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

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

* pre3
  THM >>∀w: WORDS. tl(w[k+1..lg]) = tl(w[k+1..lg]) in WORDS

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

* pre5
  THM >>∀a: SYMBOLS. a[|w|+1..lg] = a in WORDS

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

# selcon1
THM >>∀u,v: WORDS. ∀i: int. 0<i & i<|u| => (u+v)[i+1..lg] = (u[i+1..lg]+v) in WORDS

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

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

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

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

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

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

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

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

# suf2
THM >>∀w: WORDS. ∀i: N. w[i+1..lg] = w[1..lg]*w[1+1] in WORDS

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

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

# suf4
THM >>∀w: WORDS. ∀i: int. i<1 => w[1..|w|](i) = w(i) in SYMBOLS

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

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

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

# range3
THM >>∀w: WORDS. ∀i: (1,...,|w|). ∀r: int. 1≤r
  => w[1+1..r] = w(1..w[1+1..r]) in WORDS
* range4
  THM >> \forall w:WORDS. \forall l,r:int. 0 \leq l \& l < r
  \Rightarrow w[l+1..r] = w[l+1..r-1]*w(r) in WORDS

* range5
  THM >> \forall w:WORDS. \forall r:N. w[r+1..] = \epsilon in WORDS

* range6
  THM >> \forall w:WORDS. \forall l,r,k:int. 0 \leq l \& l < r \& 0 \leq k
  \Rightarrow (w[l+1..r]*w[r+1..r+k]) = w[l+1..r+k] in WORDS

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

* range8
  THM >> \forall w:WORDS. \forall l,r,i:N. l \leq r \Rightarrow w[l+1..r](i) = w(l+i) in SYMBOLS

* range9
  THM >> \forall w:WORDS. \forall l,r,i:int. 0 \leq l \& l < r
  \Rightarrow (w[l+1..r]*w[l+1..r]) = w[l+1..r] in WORDS

* range10
  THM >> \forall w:WORDS. \forall l,r,i:int. 0 \leq l \& l < r \& r \leq |w|
  \Rightarrow w = ((w[l+1..r]*w[l+1..r])\star w[r+1..|w|]) in WORDS

* cl15
  DEF 1.5: Induction

---

* induction_thm
  THM >> \forall P:WORDS->\mathbb{U}10. (\forall v:WORDS. P(v)) \leftrightarrow (\forall u:WORDS. P(\#u))

* ind_principle
  THM >> \forall P:WORDS->\mathbb{U}10
  \& (P(\#e) \& P(\#b):SYMBOLS. \forall z:WORDS. P(z) \rightarrow P(z+b)) \rightarrow \forall w:WORDS. P(w)

* induction_thml
  THM >> \forall P:WORDS->\mathbb{U}10,
  \forall i:int. \forall w:WORDS. |w| = i in int \Rightarrow P(w)
  \rightarrow \forall w:WORDS. P(w)

* ind_principle1
  THM >> \forall P:WORDS->\mathbb{U}10
  \& (P(\#e) \& \forall i:int. 0 < i \Rightarrow \forall z:WORDS. |z| = (i-1) in int \Rightarrow P(z)
  \rightarrow \forall z:WORDS. |z| = i in int \Rightarrow P(z)
  \rightarrow \forall w:WORDS. P(w)

* ind_principle2
  THM >> \forall P:WORDS->\mathbb{U}10
  \& (P(\#e) \& \forall i:int. 0 < i \Rightarrow \forall v:WORDS. |v| < i \Rightarrow P(v)
  \rightarrow \forall u:WORDS. |u| = i in int \Rightarrow P(u)
  \rightarrow \forall w:WORDS. P(w)

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

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

* trk1
  THM >> \forall A,B:TYPE. \forall g:A->B. \forall h:(B#SYMBOLS)->B. \forall x:A
  \rightarrow \forall g,h:(x,\epsilon) = g(x) in B

* trk2
  THM >> \forall A,B:TYPE. \forall g:A->B. \forall h:(B#SYMBOLS)->B. \forall x:A
  \rightarrow \forall w:WORDS. \forall v:SYMBOLS. \forall g:A
  \rightarrow \forall h:(TRK(g,h)(x,w),a) in B

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

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

* trk4
  THM >> \forall B:TYPE. \forall h:(B#SYMBOLS)->B. \forall u,v:WORDS. \forall x:B
  \rightarrow h*(x,(u+v)) = h*(h*(x,u),v) in B
* c2
DEF

CHAPTER 2: FINITE AUTOMATA

* c21
DEF

2.1: Basic definitions

* states
DEF STATES

* dfa
DEF DFA

* delstar
DEF <QxSYMBOLS->Q>*

* delstartype
THM >>VQ:STATES. ∀d:((Q)→(Q)). d* in ((Q)→(Q)) → (Q)

* delprop
THM >>VQ:STATES. ∀d:((Q)→(Q)). ∀q:(Q). d*(q,e) = q in (Q)

* delprop0
THM >>VQ:STATES. ∀d:((Q)→(Q)). ∀q:(Q). ∀w:WORDS. ∀a:SYMBOLS
. d(d*(q,d(a))) = d(d*(q,w),a) in (Q)

* delprop1
THM >>VQ:STATES. ∀d:((Q)→(Q)). ∀q,w:WORDS. ∀a:SYMBOLS
. d(d*(q,d(a))) = d(d*(q,v),w) in (Q)

* accepted
DEF <WORDS> ∈ L(<DFA>)

* acceptedtype
THM >>V:WORDS. ∀w:WORDS. w ∈ L(M) in UI

* accept
DEF L(<DFA>)

* load-all
ML

* c3
DEF

CHAPTER 3: PROPERTIES OF REGULAR SETS

* c31
DEF

3.1: PUMPING LEMMA

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

* pumping
THM >>∀M:DFA. ∃n:int. ∀z:WORDS. z ∈ L(M) & n≤|z| → ∃u,v,w:WORDS
. z=((u*v)*w) in WORDS & |(u*v)|≤n & 0 < |v| & ∀k:int. ((u*(v+k))*w) ∈ 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 (infix concat)
three_match : (term#term#tok list) -> term list

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

basic term definitions

INT : term - short for "make_int_term"
NIL : term - short for "make_nil_term"
VOID : term - short for "make_void_term"
mvar : tok -> term - short for "make_var_term"
imp : term - dummy term
and1 : term - dummy term
andr : term - dummy term
zero : term - PRL "0"
one : term - PRL "1"
U1 : term - PRL "U1"
U2 : term - PRL "U2"

eplfunctions

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

is_alpha_convertible : term -> term -> bool

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

first_exp : proof -> term if conclusion is a=b in T return a

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

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

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 : term -> proof -> term
typeof returns the type of an expression using special_type_information first, information from
the hypotheses of the proof and information from the structure of the expression.
- special_type_information must fail if it does not succeed
- default is (\exp.\proof. fail) as in type_of
match_subterms : term -> term -> proof -> term list
matches a term s against subterms of a given term t and returns an instantiation list for variables in t. Tries to find additional information on instances in the given proof if instance_list is incompe.
EXAMPLE:
\[ t = (x1:A1 \to A2) \to ((x3:A3 \to B \& B') \# (y3:C3 \to C4)) \]
\[ s = B'[a1,a3/x1,x3] \]
returns a1.imp.and1.a3.andr

Needs a lot of helpfunctions and the following lists

- **match_list** \[ \ldots(x_i,a_i)\ldots \] generated by match, maybe unordered or incomplete
- **elim_list** \[ \ldots(x_i,A_i)\ldots \] generated during decomposition of t, ordered during build (a helpfunction) complete from the i-th element
- **inst_list** \[ a_i.a_i+1\ldots \] instances for the xi (resp. imp. and1. 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 :prlml> tactics-1 (-2/ -3/ -4)

   Elime : 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_elim = int -> int -> tactic
   PRL-rule "intro"
   PRL-rule "elim" hyp level for sets
Hyp   = int -> tactic
Last_hyp = tactic
Last_1_hyp = tactic
use last hypothesis
Last_2_hyp = tactic
use hypothesis before the last one
CUMULATIVITY = int -> tactic
   PRL-rule "cumulativity" level
Cumulativity = tactic
   PRL-rule "cumulativity at UI"
Thinning = int_list -> tactic
   PRL-rule "thinning"
ThinLast = int -> tactic
   thinning of the last n hypotheses
Thin_last = tactic
   thinning of the last hypothesis
equal = tactic
   PRL-rule "equality"
Equal = tactic
   TRY equal
Arith = tactic
   PRL-rule "arith at UI"
   equal ORELSE (COMPLETE Arith)
Decision = tactic
Lemma = tok -> tactic
   PRL-rule "lemma" name
Seq  = term list -> tactic
   PRL-rule "seq"
Extensionality = tactic
   PRL-rule "extensionality"
FAILTAC = tactic
   fails always
Int_minus = tactic
   rule integer_equality_minus
Int_add = tactic
   rule integer_equality_addition
Int_sub = tactic
   rule integer_equality_subtraction
Int_mod = tactic
   rule integer_equality_multiplication
Int_number = tactic
   rule integer_equality_modulo
   rule integer_equality_natural_number
UI_equality = tactic
   rule equal_equality
UIUnitac = tactic
   rule universe_equality
ProdUnitac = tactic
   rule product_equality_independent
FuncUnitac = tactic
   rule function_equality_independent
Int_computation = tok -> int -> tactic
   rule integer_computation_kind no
Inteq_computation = int -> bool -> tactic
   rule int_eq_computation no bool
List_computation = int -> tactic
   rule list_computation no
Int_elim = int -> tactic
   PRL-rule "elim" hyp for integers (induction)
Direct_computation = term -> tactics
   rule direct_computation term
Hyp_computation = int -> term -> tactic
   rule direct_computation_hyp hyp term
Trivial_types = tactic
   a collection of all the Intro-rules which don't leave subgoal:
omputation tactics

\[ \begin{align*}
H \gg \text{redex} & \gg \text{contractum} \\
H \gg \text{redex in } T & \gg \text{contractum in } T \\
H \gg \text{redex = t in } T & \gg \text{contractum = t in } T \\
H \gg t = \text{redex in } T & \gg t = \text{contractum in } T \\
H \gg t \in \text{redex} & \gg t \in \text{contractum} \\
H \gg y : \text{redex} \rightarrow T & \gg y : \text{contractum} \rightarrow T \\
H \gg y : T \rightarrow \text{redex} & \gg y : T \rightarrow \text{contractum} \\
H \gg t \in y : T \rightarrow \text{redex} & \gg t \in y : T \rightarrow \text{contractum} \\
H \gg t \in y : \text{redex} \rightarrow T & \gg t \in y : \text{contractum} \rightarrow T
\end{align*} \]

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

The same for hypotheses:

\[ \begin{align*}
\text{Hyp compute} & \text{ hyp howoften} \\
\text{Hyp compute term} & \text{ hyp howoften} \\
\text{Hyp compute snd term} & \text{ hyp howoften} \\
\text{Hyp compute type} & \text{ hyp howoften}
\end{align*} \]

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

Elementary tacticals

\[ \begin{align*}
\text{THEMEMBER} & = \text{tactic} \rightarrow \text{tactic} \rightarrow \text{tactic} \\
\text{REPEATL} & = (\text{tactic} \rightarrow \text{tactic}) \rightarrow \text{tactic list} \rightarrow \text{tactic} \\
\text{apply tac2 to all membership subgoals of tac1} \\
\text{repeat application of the tactical to all tactics in the tactic list}
\end{align*} \]

Hypothesis elimination

\[ \begin{align*}
\text{elim using} : & \text{tactic} \rightarrow \text{term list} \rightarrow \text{int} \rightarrow \text{tactic} \\
\text{eliminate using} : & \text{term list} \rightarrow \text{int} \rightarrow \text{tactic}
\end{align*} \]

Given an instantiation list of the form \([a_1:a_2:...:a_n]\) where ai might be either instance term or one of the special terms "imp", "and1" or "andr"

\text{elim using} eliminates the named hypothesis hyp corresponding to the ai's
- nulltac will be used if inst_list is empty
- subgoals that the ai are of the right type are left over

Example:

\[ \begin{align*}
\text{hypothesis hyp be} & = "x1:T1 \rightarrow x2:T2 \rightarrow (x3:T3 \rightarrow T4 \& T4') \& (y3:T3' \rightarrow T4'')" \\
\text{and inst_list} & = [a:imp; and1; b:andr]
\end{align*} \]

Result \( H \gg T \) by \text{elim using} ..... 

\[ \begin{align*}
\gg a \in T1 \\
\gg T2 \\
\gg b \in T3 \\
T4'[a,b/x1,x3] > T
\end{align*} \]
xtended hypothesis tactic

Direct_hypotheses = tactic
Hyp_elim_on = term list -> int -> tactic
HYP = int -> tactic

Try_elimination_in_hypotheses = tactic

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

membership tactic

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

member_tac : tactic -> tactic;

member : tactic

Intended to solve all kinds 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 FAIL_TAC 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 x1:T1. \forall x2:T2. (T3 => (( \forall x4:T4. T5) & (S4 => S5)) & S3

THEOREM1 needs a given instantiation list and produces

H >> T by THEOREM1 name [t1;t2;and1;imp;and1;t4]
  >> t1 in T1
  >> t2 in T2[t1/x1]
  >> T3[t1,t2/x1,x2]
  >> t4 in T4[t1,t2/x1,x2]
  >> T5[t1,t2,t4/x1,x2,x4] >> T

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

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

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

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

f and & are defined as in the logic section - the implementation of THEOREM however is independent of this file.
. Constructors, Destructors & Predicates

a: Constructors

```plaintext
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/FALSE = term
make_not_term  = term -> term
make_all1_term = term -> term -> term -> term
make_all2_term = term -> term -> term -> term -> term
make_all3_term = term -> term -> term -> term -> term -> term
make_some1_term = term -> term -> term -> term
make_some2_term = term -> term -> term -> term -> term
make_some3_term = term -> term -> term -> term -> term -> term
```

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

b: Destructors

split the constructions named above into their parts. The result is a product term.

e.g. destruct_all2 = term -> (term#term#term#term) : "\forall x,y,type,term" into x,y,type,term

The names of these destructors are obvious.

c: Predicates

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

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

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

Ia: Formation

\[ H \vdash A \& B \text{ in } \mathcal{U} \text{ by } \text{intro} \]
\[ \quad \text{and}_{-\text{intro}} \]
\[ H \vdash A \mid B \text{ in } \mathcal{U} \text{ by } \text{intro} \]
\[ \quad \text{or}_{-\text{intro}} \]
\[ H \vdash A \rightarrow B \text{ in } \mathcal{U} \text{ by } \text{intro} \]
\[ \quad \text{imp}_{-\text{intro}} \]
\[ H \vdash A \leftrightarrow B \text{ in } \mathcal{U} \text{ by } \text{intro} \]
\[ \quad \text{equiv}_{-\text{intro}} \]
\[ H \vdash \text{false} \text{ in } \mathcal{U} \text{ by } \text{intro} \]
\[ \quad \text{false}_{-\text{intro}} \]
\[ H \vdash \neg A \text{ in } \mathcal{U} \text{ by } \text{intro} \]
\[ \quad \text{not}_{-\text{intro}} \]

\[ H \vdash (\forall x : A . B) \text{ in } \mathcal{U} \text{ by } \text{intro} \]
\[ \quad \text{all}_{-\text{intro}} \]

\[ H \vdash (\exists x_1 : A_1 . \ldots . x_n : A_n . B) \text{ in } \mathcal{U} \text{ by } \text{intro} \] 
\[ \quad \text{some}_{-\text{intro}} \]

\[ H \vdash (\exists x_1 : A_1 . \ldots . x_n : A_n . B) \text{ in } \mathcal{U} \text{ by } \text{intro} \] 
\[ \quad \text{repeat}_{-\text{intro}} \]
Ib: Introduction

\[
H \gg \text{and}_\equiv \quad \text{by intro}
\]
\[
H \gg \text{or}_\left\downarrow \right\downarrow \quad \text{by intro left i}
\]
\[
H \gg \text{or}_\rightarrow \quad \text{by intro right i}
\]
\[
H \gg \text{imp}_\rightarrow \quad \text{by intro i}
\]
\[
H \gg \text{equiv}_\rightarrow \quad \text{by intro i}
\]
\[
H \gg \text{false}_\rightarrow \quad \text{by intro}
\]
\[
H \gg \text{not}_\rightarrow \quad \text{by intro i}
\]
\[
H \gg \text{all}_\rightarrow \quad \text{by intro i}
\]
\[
H \gg \text{repeat}_\rightarrow \quad \text{by intro i \ [n]}
\]
\[
H \gg \text{some}_\rightarrow \quad \text{by intro i \ a}
\]
\[
H \gg \text{repeat}_\rightarrow \quad \text{by intro i \ a1..an \ [n]}
\]
Ic: Elimination
-------------
\[ H, \text{A1} \quad \& \quad \text{A2} \ldots \& \text{An}, H' \rightarrow T \]
\[ H, H', \text{A1}, \text{A2}, \ldots, \text{An} \rightarrow T \]
\[ H, \text{A1} \mid \text{A2} \ldots | \text{An}, H' \rightarrow T \]
\[ \text{A1} \rightarrow T \]
\[ \text{A2} \rightarrow T \]
\[ \ldots \]
\[ \text{An} \rightarrow T \]
\[ H, \text{A} \rightarrow \text{B}, H' \rightarrow T \]
\[ \rightarrow \text{A} \]
\[ \text{B} \rightarrow T \]
\[ H, \text{A} \leftrightarrow \text{B}, H' \rightarrow T \]
\[ \rightarrow \text{A} \]
\[ \text{B} \rightarrow T \]
\[ H, \text{A} \leftrightarrow \text{B}, H' \rightarrow T \]
\[ \rightarrow \text{B} \]
\[ \text{A} \rightarrow T \]
\[ H, \text{FALSE}, H' \rightarrow T \]
\[ H, \text{not} \ 	ext{A}, H' \rightarrow T \]
\[ H, H' \rightarrow \text{A} \]
\[ H, \forall x_1 : \text{A1} \ldots \forall x_n : \text{An} . B, H' \rightarrow T \]
\[ \rightarrow \text{a1} \in \text{A1} \]
\[ \ldots \]
\[ \rightarrow \text{an} \in \text{An}[\text{a1}, \ldots, \text{an-1}/x_1, \ldots, x_n-1] \]
\[ B[\text{a1}, \ldots, \text{an}/x_1, \ldots, x_n] \rightarrow T \]
\[ H, \exists x_1 : \text{A1} \ldots \exists x_n : \text{An} . B, H' \rightarrow T \]
\[ H, H', \text{x1}:\text{A1}, \ldots, \text{xn}:\text{An}, \text{B} \rightarrow T \]

Ic: Computation
---------------
Computation for logic expressions
II. Tactics

\[ H \mathrel{\bowtie} \forall x : A . B \]
\[ x : A \mathrel{\bowtie} B \]

by \texttt{AllIntro j AUjtac}

\[ H \mathrel{\bowtie} \forall x_1 : A_1 \ldots x_n : A_n . B \]
\[ x_1 : A_1, \ldots, x_n : A_n \mathrel{\bowtie} B \]

by \texttt{RepeatAllIntro j AiUjtacs}

\[ H \mathrel{\bowtie} \exists y : A . B \]
\[ B[a/y] \]

by \texttt{SomeIntro j a atac BUjtac}

\[ H \mathrel{\bowtie} \exists y_1 : A_1 \ldots \exists y_n : A_n . B \]
\[ B[a_1, \ldots, an/y_1, \ldots, yn] \]

by \texttt{RepeatSomeIntro j a list aitacs AiUjtacs BUjtac}

\[ H \mathrel{\bowtie} A \rightarrow B \]
\[ A \mathrel{\bowtie} B \]

by \texttt{ImpIntro j AUjtac}

\[ H \mathrel{\bowtie} A \leftrightarrow B \]
\[ A \mathrel{\bowtie} B \]
\[ B \mathrel{\bowtie} A \]

by \texttt{EquivIntro j AUjtac BUjtac}

\texttt{aitacs} must prove \"\texttt{\bowtie\bowtie} ai \texttt{in} Ai[a_1, \ldots, ai-1/x_1, \ldots, xi-1]\"\)
\texttt{AiUjtacs} \"\texttt{\bowtie\bowtie} Ai \texttt{in} Uj\"
\texttt{BUjtac} \"\texttt{\bowtie\bowtie} B \texttt{in} Uj\"

Using \texttt{IDTAC} instead leaves the corresponding subgoals open

\textbf{Short forms:}

\texttt{level j = 1}

\begin{itemize}
  \item \texttt{All\_intro AUjtac}
  \item \texttt{Repeat\_all\_intro AiUjtacs}
  \item \texttt{Some\_intro a atac BUjtac}
  \item \texttt{Repeat\_some\_intro allist aitacs AiUjtacs BUjtac}
  \item \texttt{Imp\_intro AUjtac}
  \item \texttt{Equiv\_intro AUjtac BUjtac}
\end{itemize}

\texttt{level j = 1 and AUjtac = membership (+BUjtac = membership)}

\begin{itemize}
  \item \texttt{Allintro}
  \item \texttt{Impintro}
  \item \texttt{Equivintro}
\end{itemize}

\textbf{to be written:}

\begin{itemize}
  \item \texttt{Repeat\_and\_equality}
  \item \texttt{Repeat\_and\_equality\_for n}
  \item \texttt{Repeat\_and\_intro}
  \item \texttt{Repeat\_and\_intro\_for n}
\end{itemize}

\textbf{II\-introduction for standard PRL\-types}

\[ H \mathrel{\bowtie} \forall x : \text{int} . T \]
\[ x : \text{int} \mathrel{\bowtie} T \]

by \texttt{int\_all\_intro}

\[ H \mathrel{\bowtie} \forall x : \text{int} . T \]
\[ x : \text{int}, k : \text{int}, k < 0, \text{indhyp: } T[k+1/x] \mathrel{\bowtie} T[k/x] \]
\[ x : \text{int} \mathrel{\bowtie} T[0/x] \]
\[ x : \text{int}, k : \text{int}, 0 < k, \text{indhyp: } T[k-1/x] \mathrel{\bowtie} T[k/x] \]

\texttt{to be written}

\begin{itemize}
  \item \texttt{void\_all\_intro}
  \item \texttt{atom\_all\_intro}
\end{itemize}
Constructors, Deststructors & Predicates

a: Constructors

\[
\begin{align*}
\text{make\_tup2\_term} & : \text{term} \to \text{term} \to \text{term} \quad \text{a b} \quad \text{into} \quad "(a, b)" \\
\text{make\_tup3\_term} & : \text{term} \to \text{term} \to \text{term} \to \text{term} \quad \text{a b c} \quad \text{into} \quad "(a, b, c)" \\
\text{make\_tup4\_term} & : \text{term} \to \text{term} \to \text{term} \to \text{term} \to \text{term} \quad \text{a b c d} \quad \text{into} \quad "(a, b, c, d)" \\
\text{make\_where3\_term} & : \text{term} \to \text{term} \to \text{term} \to \text{term} \to \text{term} \quad \text{a b c t'} \quad \text{into} \quad "t - where \ (a, b) = t'" \\
\text{make\_where4\_term} & : \text{term} \to \text{term} \to \text{term} \to \text{term} \to \text{term} \to \text{term} \quad \text{t a b c d t'} \quad \text{into} \quad "t - where \ (a, b, c, d) = t'" \\
\text{make\_id\_function\_term/id} & : \text{term} \to \text{term} \quad \text{t} \quad \text{into} \quad \"\lambda x.x\" \\
\text{make\_lambda2\_term} & : \text{term} \to \text{term} \to \text{term} \to \text{term} \quad \text{x y t} \quad \text{into} \quad \"\lambda x.y.t\" \\
\text{make\_apply2\_term} & : \text{term} \to \text{term} \to \text{term} \to \text{term} \quad \text{f a b} \quad \text{into} \quad \"f(a, b)\" \\
\text{additional\ constructors} \\
\text{make\_equal\_int\_term} & : \text{term} \text{ list} \to \text{term} \quad \text{[x:y:..]} \quad \text{into} \quad \"x = y..\ in\ int\"
\end{align*}
\]

I. Rules

-----

Ia: Formation

--- No formation rules for tupleing

Ib: Introduction

\[
\begin{align*}
\text{H} & \gg (a_1, a_2, \ldots, a_n) \quad \text{in} \quad x_1 : A_1 # x_2 : A_2 # \ldots # A_n \\
& \gg a_1 \quad \text{in} \quad A_1 \\
& \gg a_2 \quad \text{in} \quad A_2[a_1/x_1] \\
& \quad \ldots \\
& \gg a_n \quad \text{in} \quad A_n \quad \text{[a_1, \ldots, a_{n-1}/ x_1, \ldots, x_{n-1}]} \\
& \quad x_1 : A_1, \ldots, x_{n-1} : A_{n-1} \quad \text{[A in Ui]} \\
& \quad x_1 : A_1 \gg a_2 \quad \text{in} \quad \text{Ui} \\
\end{align*}
\]

\[\text{H} \gg t -\ where\ (a, b) = x - \ in \ T\]
\[\gg x \text{ in} \ y : A # B\]
\[\quad \text{a_1 : A, b_1 : B[a/y], x=(a_1, b_1) \ in \ y : A # B \ >> \ t[a_1, b_1/a, b] \ in \ T}\]

\[\text{H} \gg t -\ where\ (a, b, c) = x - \ in \ T\]
\[\gg x \text{ in} \ y_1 : A_1 # y_2 : A_2 # A_3\]
\[\quad y_1 : A_1 \gg y_2 : A_2 # A_3 \quad \text{in} \quad \text{Ui}\]
\[\quad \text{a_1 : A, b_1 : A_2[a/y_1], c_1 : A_3[a/b_1/y_1, y_2], x=(a_1, b_1, c_1) \ in} \ y_1 : A_1 # y_2 : A_2 # A_3\]
\[\gg t[a_1, b_1, c_1/a, b, c] \quad \text{in} \quad \text{T}\]

\[\text{H} \gg t -\ where\ (a, b, c, d) = x - \ in \ T\]
\[\gg x \text{ in} \ y_1 : A_1 # y_2 : A_2 # y_3 : A_3 # A_4\]
\[\quad y_1 : A_1, y_2 : A_2 \gg y_3 : A_3 # A_4 \quad \text{in} \quad \text{Ui}\]
\[\quad y_1 : A_1 \gg y_2 : A_2 # y_3 : A_3 # A_4 \quad \text{in} \quad \text{Ui}\]
\[\quad \text{a_1 : A, b_1 : A_2[a_1/y_1], c_1 : A_3[a_1/b_1/y_1, y_2], d_1 : A_4[a_1/b_1/c_1/y_1, y_2, y_3]\]
\[\quad x=(a_1, b_1, c_1, d_1) \quad \text{in} \ y_1 : A_1 # y_2 : A_2 # y_3 : A_3 # A_4\]
\[\gg t[a_1, b_1, c_1, d_1/a, b, c, d] \quad \text{in} \quad \text{T}\]

\[\text{H} \gg \lambda a, b, t \quad \text{in} \ y : (x : A # B) \to C\]
\[\quad a : A, b : B[a/x] \quad \text{t in} \ C\]
\[\gg x : A # B \quad \text{in} \quad \text{Ui}\]

\[\text{H} \gg f(a, b) \quad \text{in} \ C\]
\[\gg f \text{ in} \ y : (x : A # B) \to C\]
\[\gg a \quad \text{in} \ A\]
\[\gg b \quad \text{in} \ B[a/x]\]
\[\gg x : A \quad \text{in} \quad \text{Ui}\]
IC: Elimination

Id: Computation

\[ H \gg t - \text{where} (a_1, \ldots, a_n) = (b_1, \ldots, b_n) = t' \text{ in } T \text{ by } \text{treduce 1} \]
\[ \gg t[b_1, \ldots, b_n/a_1, \ldots, a_n] = t' \text{ in } T \]

\[ H \gg (\lambda x, y.t)(a, b) = t' \text{ in } T \text{ by } \text{treduce 1} \]
\[ \gg t[a, b/x, y] = t' \text{ in } T \]

\[ H \gg t - \text{where} (a_1, \ldots, a_n) = (b_1, \ldots, b_n) \text{ by } \text{tcompute} \]
\[ \gg t[b_1, \ldots, b_n/a_1, \ldots, a_n] \]

\[ H \gg (\lambda x, y.t)(a, b) \text{ by } \text{tcompute} \]
\[ \gg t[a, b/x, y] \]

\[ H, t - \text{where} (a_1, \ldots, a_n) = (b_1, \ldots, b_n) = t' \text{ in } T, H' \gg T' \]
\[ H, t[b_1, \ldots, b_n/a_1, \ldots, a_n] = t' \text{ in } T, H' \gg T' \]

\[ H, t - \text{where} (a_1, \ldots, a_n) = (b_1, \ldots, b_n) = t' \text{ in } T, H' \gg T' \]
\[ H, t[b_1, \ldots, b_n/a_1, \ldots, a_n], H' \gg T' \]

\[ H, (\lambda x, y.t)(a, b) = t' \text{ in } T, H' \gg T' \]
\[ H, t[a, b/x, y] = t' \text{ in } T, H' \gg T' \]

I. Tactics

COMPUTATION

\[ H \gg f(x, \text{redex}) [= t] \text{ in } T \]
\[ \gg f(x, \text{contractum}) [= t] \text{ in } T \]

\[ H \gg t = f(x, \text{redex}) \text{ in } T \]
\[ \gg t = f(x, \text{contractum}) \text{ in } T \]

\[ H \gg f(\text{redex}, w) [= t] \text{ in } T \]
\[ \gg f(\text{contractum}, w) [= t] \text{ in } T \]

\[ H \gg t = f(\text{redex}, w) \text{ in } T \]
\[ \gg t = f(\text{contractum}, w) \text{ in } T \]

\[ H \gg \text{redex}(x, w) [= t] \text{ in } T \]
\[ \gg \text{contractum}(x, w) [= t] \text{ in } T \]

\[ H \gg t = \text{redex}(x, w) \text{ in } T \]
\[ \gg t = \text{contractum}(x, w) \text{ in } T \]

Compute_apply2_arg2
Compute_snd_apply2_arg2
Compute_apply2_arg1
Compute_snd_apply2_arg1
Compute_apply2_fun
Compute_snd_apply2_fun
. Term constructors, predicates and destructors

a: Constructors

make_SETS_term/SETS = term
make_carrier_term = term -> term
make_pred_term = term -> term
make_setdef_term = term -> term -> term -> term
make_singleton_term = term -> term -> term
make_inset_term = term -> term
make_set_conversion = term -> term
make_powerset_term = term -> term

S into "carrier(S)"
S into "pred(S)"
{x:A} into "(x:A)"
x A B into "{x:A||B}"
x S into "x c S"
S into "(S)"
A into "P(A)"

I. RULES

help tactic
SETS_spread = tactic

Ia: Formation

H |> SETS in U1 by Sintro (i>1) SETS_equality / SETSeq
H |> {S} in U1 by Sintro
   |> S in SETS
   SETS_equality_conversion / Sconversion
H |> P(A) in U1 by Sintro (i>1)
   |> A in TYPE
   SETS_equality_power / Spower
H |> x ∈ (S) in U1 by Sintro
   |> S in SETS
   |> x in carrier(S)
   SETS_equality_inset / Sinset

Ib: Introduction

H |> (x,p) in SETS by Sintro
   |> X in TYPE
   |> P:X -> U1
   SETS_equality_pair / Spair
H |> S in P(A) by Sintro
   |> S in SETS
   |> carrier(S) = A in U1
   SETS_power_element / Spowerel
H |> x in (S) by Sintro
   |> x in carrier(S)
   |> pred(S)(x)
   SETS_conversion_element / Sconvel
H |> carrier(S) in U1 by Sintro
   |> S in SETS
   SETS_equality_carrier / Scarrier
H |> pred(S) in carrier(S)→U1 by Sintro
   |> S in SETS
   SETS_equality_pred / Spred
Ic: Elimination

\[ H, S \in \text{SETS}, H' \gg T \quad \text{SETS}_{\text{elim}} \text{ hyp} \]
\[ \quad \text{carrier: TYPE, pred: carrier} \rightarrow \text{U1, } S = (\text{carrier, pred}) \in \text{SETS} \]
\[ \quad \gg T[(\text{carrier, pred})/ S] \]

\[ H, x \in (S), H' \gg T \quad \text{conversion}_{\text{elim}} \text{ hyp} \]
\[ \quad x \in \text{carrier}(S), x \notin \{S\} \gg T \]

\[ H, S \in \text{P(A)}, H' \gg T \quad \text{powerset}_{\text{elim}} \text{ hyp} \]
\[ \quad \gg A \in \text{TYPE} \]
\[ \quad \text{pred: A} \rightarrow \text{U1, } S = (A, \text{pred}) \in \text{SETS} \gg T[(A, \text{pred})/ S] \]

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

Id: Computation

II: Miscellaneous functions and tactics

\text{sets}_{\text{all intro}} = \text{tactic} \quad \text{all-introduction for SETS}
finset.ml: PRL-extensions for natural numbers/ arithmetic/ finite sets etc.

 Constructors, Destructors & Predicates

a: Constructors

- `make_leq_term : term -> term -> term`  
  `i j` into "i≤j"

- `make_uneq_term : term -> term -> term`  
  `i j` into "i≠j"

- `make_nat_term/nat : term`  
  "N" into "{1,...,j}"

- `make_nbar1_term : term`  
  "{0,...,j}" into "FINITE SETS"

- `make_finite_term/finite_sets : term`  
  "A" into "{A}"

Destructors and predicates as usual

I. Rules

(nearly all of these rules aren't written yet)

Ia: Formation

- `H >> isj in Ui by fintro`  
  leq_intro

- `H >> i≠j in Ui by fintro`  
  uneq_intro

- `H >> N in Ui by fintro`  
  nat_intro

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

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

- `H >> FINITE_SETS in Ui by fintro`  
  finset_intro

- `H >> (A) in Ui by fintro`  
  finset Equality_conversion

Ib: Introduction

Ic: Elimination

Id: Computation

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

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

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

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

- `H >> 0 < j - i`  
  `i < j`
Term constructors, predicates and destructors

---

a: Constructors

---

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

```
make_symbols_term/symbols = term : "SYMBOLS"
make_words_term/words = term : "WORDS"
make_noteps_term = term -> term : w into "#w"
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 "lg(w)"
make_cutprefix_term = term -> term -> term : w i into "w[1..i]"
make_select_term = term -> term -> term : w i into "w[i]"
make_cutsuffix_term = term -> term -> term : w i into "w[i+1..j]"
make_range_term = term -> term -> term -> term : w i j into "w[1..j]"
```

? additional constructors which are useful in rules and tactics

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

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

b: Destructors

---

split the constructions named above into their parts. The result is a product term.

e.g. destruct_concat = term -> (term*term) : "(u*v)" into u,v.

The names of these destructors are obvious.

c: Predicates

---

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

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

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

Ia: Formation
-------------

- \( H \gg \text{SYMBOLS in } U_i \) by wintro \( \text{symbol}_\text{equality} \) / \( \text{sequal} \)
- \( H \gg \text{WORDS in } U_i \) by wintro \( \text{word}_\text{equality} \) / \( \text{wequal} \)
- \( H \gg \text{w\&e in } U_i \)
  \( \gg w \text{ in } \text{WORDS} \)
  by wintro \( \text{noteqs}_\text{equality} \) / \( \text{noteqs} \)

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

- \( H \gg e \text{ in } \text{WORDS} \) by wintro \( \text{word}_\text{equality}_\text{eps} \) / \( \text{weps} \)
- \( H \gg a\cdot v \text{ in } \text{WORDS} \)
  \( \gg a \text{ in } \text{SYMBOLS} \)
  \( \gg v \text{ in } \text{WORDS} \)
  by wintro \( \text{word}_\text{equality}_\text{cons} \) / \( \text{wcons} \)
- \( H \gg u\cdot v \text{ in } \text{WORDS} \)
  \( \gg u \text{ in } \text{WORDS} \)
  \( \gg v \text{ in } \text{WORDS} \)
  by wintro \( \text{word}_\text{equality}_\text{concat} \) / \( \text{wconcat} \)
- \( H \gg @a \text{ in } \text{WORDS} \)
  \( \gg a \text{ in } \text{SYMBOLS} \)
  by wintro \( \text{word}_\text{equality}_\text{symbols} \) / \( \text{wsym} \)
- \( H \gg u\cdot a \text{ in } \text{WORDS} \)
  \( \gg u \text{ in } \text{WORDS} \)
  \( \gg a \text{ in } \text{SYMBOLS} \)
  by wintro \( \text{word}_\text{equality}_\text{antiscons} \) / \( \text{wanti} \)
- \( H \gg +w \text{ in } \text{WORDS} \)
  \( \gg w \text{ in } \text{WORDS} \)
  by wintro \( \text{word}_\text{equality}_\text{rev} \) / \( \text{wrev} \)
- \( H \gg w\cdot i \text{ in } \text{WORDS} \)
  \( \gg i \text{ in } \text{int} \)
  \( \gg w \text{ in } \text{WORDS} \)
  by wintro \( \text{word}_\text{equality}_\text{iter} \) / \( \text{witer} \)
- \( H \gg hd(w) \text{ in } \text{SYMBOLS} \)
  \( \gg w \text{ in } \text{WORDS} \)
  by wintro \( \text{word}_\text{equality}_\text{hd} \) / \( \text{whd} \)
- \( H \gg tl(w) \text{ in } \text{WORDS} \)
  \( \gg w \text{ in } \text{WORDS} \)
  by wintro \( \text{word}_\text{equality}_\text{tl} \) / \( \text{wtl} \)
- \( H \gg [w] \text{ in } \text{int} \)
  \( \gg w \text{ in } \text{WORDS} \)
  by wintro \( \text{word}_\text{equality}_\text{lg} \) / \( \text{wlg} \)
- \( H \gg w[j+1..lg] \text{ in } \text{WORDS} \)
  \( \gg j \text{ in } \text{int} \)
  \( \gg w \text{ in } \text{WORDS} \)
  by wintro \( \text{word}_\text{equality}_\text{cutprefix} \) / \( \text{wpre} \)
- \( H \gg w(i) \text{ in } \text{SYMBOLS} \)
  \( \gg i \text{ in } \text{int} \)
  \( \gg w \text{ in } \text{WORDS} \)
  by wintro \( \text{word}_\text{equality}_\text{select} \) / \( \text{wselect} \)
- \( H \gg w[1..i] \text{ in } \text{WORDS} \)
  \( \gg i \text{ in } \text{int} \)
  \( \gg w \text{ in } \text{WORDS} \)
  by wintro \( \text{word}_\text{equality}_\text{cutsuffix} \) / \( \text{wsuf} \)
- \( H \gg w[1+1..r] \text{ in } \text{WORDS} \)
  \( \gg l \text{ in } \text{int} \)
  \( \gg r \text{ in } \text{int} \)
  \( \gg w \text{ in } \text{WORDS} \)
  by wintro \( \text{word}_\text{equality}_\text{range} \) / \( \text{wrange} \)
\textbf{IC: Elimination} \hfill \textbf{(T uses a theorem)}

\textit{There is no SYMBOL elimination}

\begin{align*}
\text{H, w:WORDS, H'} & \gg T \\
& \gg T[e\langle w \rangle] \\
& \quad \text{hd:SYMBOLS, tl:WORDS, tl\_hyp:T[tl\langle w \rangle]} \gg T[hd.tl\langle w \rangle] & \text{\textit{word\_elim} hyp} \\
\text{H, w:WORDS, H'} & \gg T \\
& \quad u:WORDS \gg T[u\langle w \rangle] \text{ in U10} \\
& \quad \gg T[e\langle w \rangle] \\
& \quad a:SYMBOLS, v:WORDS, T[v\langle w \rangle] \gg T[v+a\langle w \rangle] & \text{\textit{word\_elim\_tail} hyp} \\
\text{H, w:WORDS, H'} & \gg T \\
& \quad u:WORDS \gg T[u\langle w \rangle] \text{ in U10} \\
& \quad \gg T[e\langle w \rangle] \\
& \quad i:\text{int, } 0<i, (\forall v:WORDS. \ |v|=i-1 \text{ in int } \Rightarrow T[v\langle w \rangle]), u:WORDS, \ |u|=i \text{ in int} \\
& \quad \gg T[u\langle w \rangle] & \text{\textit{word\_elim\_lg} hyp}
\end{align*}

\textbf{Additional Helpfunctions}

\begin{itemize}
\item \texttt{word\_equality\_induction} : tactic
\item \texttt{word\_integer\_induction} : tactic
\end{itemize}
I. RULES

(Id: Computation)

H >> \sigma v = w in WORDS by wreduce 1
  >> v = w in WORDS

H >> \sigma \sigma = w in WORDS by wreduce 1
  >> \sigma = w in WORDS

H >> \alpha \sigma v = w in WORDS by wreduce 1
  >> \alpha \cdot v = w in WORDS

H >> (\alpha u \sigma v) = w in WORDS by wreduce 1
  >> \alpha \cdot (u \sigma v) = w in WORDS

H >> (u \sigma v) \alpha = w in WORDS by wreduce 1
  >> u \cdot \sigma v \cdot \alpha = w in WORDS

H >> e = w in WORDS by wreduce 1
  >> e = w in WORDS

H >> a \sigma a = w in WORDS by wreduce 1
  >> a \sigma a = w in WORDS

H >> \alpha (a . u) = w in WORDS by wreduce 1
  >> \alpha \cdot u \sigma a = w in WORDS

H >> \eta (u \sigma v) = w in WORDS by wreduce 1
  >> \eta \cdot v \cdot u \sigma = w in WORDS

H >> \eta (u \sigma a) = w in WORDS by wreduce 1
  >> \eta \cdot a \cdot u \sigma = w in WORDS

H >> \eta (+u) = w in WORDS by wreduce 1
  >> \eta \cdot u = w in WORDS

H >> hd(a . l) = b in SYMBOLS by wreduce 1
  >> a = b in SYMBOLS

H >> tl(e) = w in WORDS by wreduce 1
  >> e = w in WORDS

H >> tl(a . v) = w in WORDS by wreduce 1
  >> v = w in WORDS

H >> |e| = x in int by wredulce1
  >> 0 = x in int

H >> |a . l| = x in int by wredulce1
  >> |l| + 1 = x in int

H >> |oa| = x in int by wredulce1
  >> 1 = x in int
\[ \text{H} \gg [a \cdot v] = x \text{ in int by \text{wreduce} 1} \]
\[ \quad \gg u \text{ in \text{WORDS}} \]
\[ \quad \gg v \text{ in \text{WORDS}} \]
\[ \quad \gg |u| + |v| = x \text{ in int} \]

\[ \text{H} \gg [u \cdot a] = x \text{ in int by \text{wreduce} 1} \]
\[ \quad \gg u \text{ in \text{WORDS}} \]
\[ \quad \gg a \text{ in \text{SYMBOLS}} \]
\[ \quad \gg |u| + 1 = x \text{ in int} \]

\[ \text{H} \gg [v \cdot v] = x \text{ in int by \text{wreduce} 1} \]
\[ \quad \gg v \text{ in \text{WORDS}} \]
\[ \quad \gg |v| = x \text{ in int} \]

\[ \text{H} \gg [\text{tl}(v)] = x \text{ in int by \text{wreduce} 1} \]
\[ \quad \gg v \text{ in \text{WORDS}} \]
\[ \quad \gg v \neq \vphantom{e} \]
\[ \quad \gg |v| - 1 = x \text{ in int} \]

\[ \text{H} \gg [v[1..|v|]] = x \text{ in int by \text{wreduce} 1} \]
\[ \quad \gg |v| - 1 = x \text{ in int} \]
\[ \quad \gg v \text{ in \text{WORDS}} \]

\[ \text{H} \gg [v[1..r]] = x \text{ in int by \text{wreduce} 1} \]
\[ \quad \gg r = x \text{ in int} \]
\[ \quad \gg v \text{ in \text{WORDS}} \]

\[ \text{H} \gg [v[1..r]] = x \text{ in int by \text{wreduce} 1} \]
\[ \quad \gg r - 1 = x \text{ in int} \]
\[ \quad \gg 1 < r + 1 \]

\[ \text{H} \gg v + m = w \text{ in \text{WORDS} by} \]
\[ \quad \gg \varepsilon = w \text{ in \text{WORDS}} \]
\[ \quad \gg m < 0 \]

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

\[ \text{H} \gg v + i = w \text{ in \text{WORDS} by \text{wreduce} 1} \]
\[ \quad \gg v + (v + i - 1) = w \text{ in \text{WORDS}} \]
\[ \quad \gg 0 < i \]

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

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

\[ \text{H} \gg w[k+1..\lg] = w \text{ in \text{WORDS} by \text{wreduce} 1} \]
\[ \quad \gg \text{tl}(v[k-1+1..\lg]) = w \text{ in \text{WORDS}} \]
\[ \quad \gg 0 < k \]

\[ \text{H} \gg w[k+1..\lg] = w \text{ in \text{WORDS}} \]
\[ \quad \gg \text{tl}(v[k+1+1..\lg]) = w \text{ in \text{WORDS}} \]
\[ \quad \gg k > 0 \]

\[ \text{H} \gg \text{tl}(v[k+1..\lg]) = w \text{ in \text{WORDS} by \text{wreduce} 1} \]
\[ \quad \gg (\text{tl}(v))[k+1..\lg] = w \text{ in \text{WORDS}} \]
\[ \quad \gg v \text{ in \text{WORDS}} \]
\[ \quad \gg k \in \text{N} \]

\[ \text{H} \gg v[|v|+1..\lg] = w \text{ in \text{WORDS} by \text{wreduce} 1} \]
\[ \quad \gg \varepsilon = w \text{ in \text{WORDS}} \]
\[ \quad \gg v \text{ in \text{WORDS}} \]
\( H \Rightarrow v(1) = b \) in SYMBOLS by \( wreduce \ 1 \\
  \Rightarrow h(\upsilon) = b \) in SYMBOLS \\
  \Rightarrow \upsilon \) in WORDS

\( H \Rightarrow (a \cdot \upsilon)(i+1) = b \) in SYMBOLS by \( wreduce \ 1 \\
  \Rightarrow v(i) = b \) in SYMBOLS \\
  \Rightarrow a \) in SYMBOLS \\
  \Rightarrow 0 < i+1

\( H \Rightarrow w[1..m] = w \) in WORDS by \\
  \Rightarrow c = w \) in WORDS \\
  \Rightarrow m < 0

\( H \Rightarrow w[1..0] = w \) in WORDS by \( wreduce \ 1 \\
  \Rightarrow c = w \) in WORDS

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

\( H \Rightarrow w[1..|w|] = w \) in WORDS by \( wreduce \ 1 \\
  \Rightarrow v = w \) in WORDS

\( H \Rightarrow w[0+1..r] = w \) in WORDS by \( wreduce \ 1 \\
  \Rightarrow v[1..r] = w \) in WORDS

\( H \Rightarrow w[1..r] = w \) in WORDS by \\
  \Rightarrow v[0+1..r] = w \) in WORDS

\( H \Rightarrow v[1..|w|] = w \) in WORDS by \( wreduce \ 1 \\
  \Rightarrow v[1+1..|w|] = w \) in WORDS

\( H \Rightarrow v[1+1..lg] = w \) in WORDS by \\
  \Rightarrow v[1+1..lg] = w \) in WORDS

\( H \Rightarrow v(1) . v[1+1..r] = w \) in WORDS by \\
  \Rightarrow v[1-1+1..r] = w \) in WORDS

\( H \Rightarrow v[1+1..r] = w \) in WORDS by \( wreduce \ 1 \\
  \Rightarrow v(1) . v[1+1..r] = w \) in WORDS

\( H \Rightarrow v[1+1..r] = w \) in WORDS by \( wreduce \ 1 \\
  \Rightarrow v[1+1..r+1] = w \) in WORDS

\( H \Rightarrow v[1+1..r+1] = w \) in WORDS by \( wreduce \ 1 \\
  \Rightarrow v[1+1..r+1] = w \) in WORDS

\( H \Rightarrow v[r+1..r] = w \) in WORDS by \( wreduce \ 1 \\
  \Rightarrow c = w \) in WORDS \\
  \Rightarrow v \) in WORDS

\( H \Rightarrow v[0+1..|v|] = w \) in WORDS by \( wreduce \ 1 \\
  \Rightarrow v = w \) in WORDS

\( H \Rightarrow v[1+1..k] \cdot v[k+1..r] = w \) in WORDS by \( wreduce \ 1 \\
  \Rightarrow v[1+1..r] = w \) in WORDS \\
  \Rightarrow k \) in int \\
  \Rightarrow k \) in int \\
  \Rightarrow r \) in int \\
  \Rightarrow 0 \leq l \\
  \Rightarrow l \leq k \\
  \Rightarrow k \leq r
CONVERSIONS

H >> \exists:WORDS. \exists:SYMBOLS. z\ast b = w in WORDS
  >> w\neq c

H >> w\neq e
  >> \exists:WORDS. \exists:SYMBOLS. z\ast b = w in WORDS

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

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

H >> 0 < |w|
  >> w\neq c

H >> w\neq e
  >> 0 < |w|

H >> w\neq c
  >> \sim (w = e in WORDS)

H >> \sim (w = e in WORDS)
  >> w\neq c

Helpfunction

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

IIa: general tactics
-------------------

\textbf{Wintro} : tactic
\hspace{1cm} repeated \texttt{word_introduction}

\textbf{Wmember} : tactic
\hspace{1cm} general membership tactic including knowledge about \texttt{WORDS}

IIb: All Introduction and Induction
----------------------------------

\texttt{H \gg \texttt{va}\texttt{:SYMBOLS.T}}
\hspace{1cm} symbol\texttt{\_all\_intro}
\hspace{1cm} \texttt{a:SYMBOLS \gg T}

\texttt{H \gg \texttt{vw}\texttt{:WORDS.T}}
\hspace{1cm} word\texttt{\_all\_intro}
\hspace{1cm} \texttt{w:WORDS \gg T}

\texttt{H \gg \texttt{vw}\texttt{:WORDS.T}}
\hspace{1cm} word\texttt{\_induction}
\hspace{1cm} \texttt{T[e/w]}
\hspace{1cm} \texttt{hd:SYMBOLS, tl:WORDS, tl\_hyp:T[tl/w] \gg T[hd.tl/w]}

\texttt{H \gg \texttt{vw}\texttt{:WORDS.T}}
\hspace{1cm} word\texttt{\_tail\_induction}
\hspace{1cm} \texttt{u:WORDS \gg T[u/w] \texttt{in U10}}
\hspace{1cm} \texttt{T[e/w]}
\hspace{1cm} \texttt{a:SYMBOLS, v:WORDS, T[v/w] \gg T[v+a/w]}

\texttt{H \gg \texttt{vw}\texttt{:WORDS.T}}
\hspace{1cm} word\texttt{\_lg\_induction}
\hspace{1cm} \texttt{u:WORDS \gg T[u/w] \texttt{in U10}}
\hspace{1cm} \texttt{T[e/w]}
\hspace{1cm} \texttt{i:int, 0\leq i, (\forall v:WORDS. \texttt{|v|=i-1 in int \Rightarrow T[v/w])}, u:WORDS, \texttt{|u|=i in int}}
\hspace{1cm} \texttt{\gg T[u/w]}

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

\texttt{H \gg \texttt{vw}\texttt{:WORDS.T}}
\hspace{1cm} tail\texttt{\_induction\_using\_tac}
\hspace{1cm} \texttt{T[e/w]}
\hspace{1cm} \texttt{a:SYMBOLS, v:WORDS, T[v/w] \gg T[v+a/w]}

\texttt{H \gg \texttt{vw}\texttt{:WORDS.T}}
\hspace{1cm} lg\texttt{\_induction\_using\_tac}
\hspace{1cm} \texttt{T[e/w]}
\hspace{1cm} \texttt{i:int, 0\leq i, (\forall v:WORDS. \texttt{|v|=i-1 in int \Rightarrow T[v/w])}, u:WORDS, \texttt{|u|=i in int}}
\hspace{1cm} \texttt{\gg T[u/w]}

IIc: Some introduction for \texttt{WORDS}
-------------------------------------

\texttt{H \gg \exists w_1,..,w_n:WORDS.T}
\hspace{1cm} \texttt{word\_some\_intro level [v_1;..;v_n]}
\hspace{1cm} \texttt{w_1,..,w_n:WORDS \gg T in U1i}
\hspace{1cm} \texttt{T[v_1,..,v_n/w_1,..,w_n]}
\hspace{1cm} ("v_i in \texttt{WORDS}" must be provable by \texttt{Wmember})

IID: Noteps Introduction
------------------------

\texttt{H \gg a.1=e}
\hspace{1cm} \texttt{word\_intro\_noteps / wnoteps}
\hspace{1cm} \texttt{a in SYMBOLS}
\hspace{1cm} \texttt{1 in WORDS}

IIe: a special variant of \texttt{con\_asoz}
----------------------------------------

\texttt{H \gg \texttt{(u*v)*w = x*(y*z) in WORDS}}
\hspace{1cm} \texttt{by con\_asoz 1}
\hspace{1cm} \texttt{u = x in WORDS}
\hspace{1cm} \texttt{v = y in WORDS}
\hspace{1cm} \texttt{w = z in WORDS}
:: Constructors, destructors & predicates

\[
\begin{align*}
\text{make_trk_term} & \quad = \quad \text{term} \to \text{term} \to \text{term} & \quad g \quad h & \quad \text{into} & \quad "\text{TRK}(g,h)" \\
\text{make_idtrk_term} & \quad = \quad \text{term} \to \text{term} & \quad h & \quad \text{into} & \quad "h*" \\
\text{make_trktype_term} & \quad = \quad \text{term} \to \text{term} \to \text{term} & \quad A \quad B & \quad \text{into} & \quad "(A#WORDS) \to B"
\end{align*}
\]

destructors & predicates as usual

I: RULES

---------

Ib: Introduction

\[
\begin{align*}
H \quad \triangleright \quad \text{TRK}(g,h) \text{ in } (A#WORDS) \to B \\
& \quad \triangleright \quad A \text{ in } \text{TYPE} \\
& \quad \triangleright \quad B \text{ in } \text{TYPE} \\
& \quad \triangleright \quad g \text{ in } A \to B \\
& \quad \triangleright \quad h \text{ in } (B#SYMBOLS) \to B  \\
\text{trk_intro}
\end{align*}
\]

\[
\begin{align*}
H \quad \triangleright \quad h* \text{ in } (B#WORDS) \to B \\
& \quad \triangleright \quad B \text{ in } \text{TYPE} \\
& \quad \triangleright \quad h \text{ in } (B#SYMBOLS) \to B  \\
\text{trkid_intro}
\end{align*}
\]

\[
\begin{align*}
H \quad \triangleright \quad h*(x,w) \text{ in } B \\
& \quad \triangleright \quad B \text{ in } \text{TYPE} \\
& \quad \triangleright \quad h \text{ in } (B#SYMBOLS) \to B \\
& \quad \triangleright \quad x \text{ in } B \\
& \quad \triangleright \quad w \text{ in } \text{WORDS}  \\
\text{trkid_equality_apply}
\end{align*}
\]

Id: Computation

\[
\begin{align*}
H \quad \triangleright \quad \text{TRK}(g,h)(x,c) = t \text{ in } B \quad \text{by trk_reduce 1} \\
& \quad \triangleright \quad g(x) = t \text{ in } B \\
\text{trk_reduce_base no}
\end{align*}
\]

\[
\begin{align*}
H \quad \triangleright \quad \text{TRK}(g,h)(x,v*e a) = t \text{ in } B \quad \text{by trk_reduce 1} \\
& \quad \triangleright \quad h(\text{TRK}(g,h)(x,v),a) = t \text{ in } B \\
\text{trk_reduce_up no}
\end{align*}
\]

\[
\begin{align*}
H \quad \triangleright \quad h*(x,c) = t \text{ in } B \quad \text{by trk_reduce 1} \\
& \quad \triangleright \quad x = t \text{ in } B  \\
\text{trk_reduce_base no}
\end{align*}
\]

\[
\begin{align*}
H \quad \triangleright \quad h*(x,v*e a) = t \text{ in } B \quad \text{by trk_reduce 1} \\
& \quad \triangleright \quad h(h*(x,v),a) = t \text{ in } B \\
& \quad \triangleright \quad v \text{ in } \text{WORDS} \\
& \quad \triangleright \quad a \text{ in } \text{SYMBOLS} \\
& \quad \triangleright \quad B \text{ in } \text{TYPE} \\
& \quad \triangleright \quad x \text{ in } B \\
& \quad \triangleright \quad h \text{ in } (B#SYMBOLS) \to B  \\
\text{trk_reduce_up no}
\end{align*}
\]

\[
\begin{align*}
H \quad \triangleright \quad h*(x,u*v) = t \text{ in } B \quad \text{by trk_reduce 1} \\
& \quad \triangleright \quad h*(h*(x,u),v) = t \text{ in } B \\
& \quad \triangleright \quad v \text{ in } \text{WORDS} \\
& \quad \triangleright \quad w \text{ in } \text{WORDS} \\
& \quad \triangleright \quad B \text{ in } \text{TYPE} \\
& \quad \triangleright \quad x \text{ in } B  \\
\text{trk_reduce_concat no}
\end{align*}
\]

I Tactics

Try to prove the unnecessary subgoals of star_reduction, dfa_equality_apply

\[
\begin{align*}
\text{trkid_anticons_compute : int -> tactic} \\
\text{trkid_concat_compute : int -> tactic} \\
\text{trkid_apply : tactic}
\end{align*}
\]
. Constructors, destructors & predicates

a: Constructors

\[
\text{make\_states\_term/\text{states}} = \text{term} \\
\text{make\_dfa\_term/\text{dfa}} = \text{term} \\
\text{make\_aut\_term} = \text{term} \to \text{term} \to \text{term} \to \text{term} \\
\text{make\_star\_term} = \text{term} \to \text{term} \\
\text{make\_accept\_term} = \text{term} \to \text{term} \\
\text{make\_accepted\_term} = \text{term} \to \text{term} \to \text{term}
\]

Q d q0 F into "\text{(Q,d,q0,F)}"

d into "d"*

M into "\text{L(M)}"

w M into "w \in \text{L(M)}"

Additional functions

\[
\text{make\_table\_type\_term} = \text{term} \to \text{term} \\
Q \to ((Q\#\text{SYMBOLS}) \to (Q))
\]

Destructors & Predicates as usual

I. RULES

(#) unfinished

Helpfunction:
H, Q:\text{STATES}, H' \gg ((Q\#\text{SYMBOLS}) \to (Q)) \# \text{Q} \# P(\text{Q}) in \text{UI} by \text{dfa\_parts\_intro} (i \geq 1)

Ia: Formation

---

H \gg \text{STATES} in \text{UI} by \text{df intro} state\_intro

H \gg \text{Q} in \text{UI} by \text{df intro} state\_set\_intro

\gg Q in \text{STATES}

H \gg \text{DFA} in \text{UI} by \text{df intro} (i \geq 1) dfa\_intro

H \gg (w \in L(M)) in \text{UI} by \text{df intro} accepted\_intro

\gg M in \text{DFA}

\gg w in \text{WORDS}

H \gg L(M) in \text{UI} by \text{df intro} accept\_intro

Ib: Introduction

---

H \gg (Q,d,q0,qf) in \text{DFA} by \text{df intro} dfa\_equality

\gg Q in \text{STATES}

\gg d in ((Q\#\text{SYMBOLS}) \to Q)

\gg q0 in \text{Q}

\gg qf in \text{Q}

H \gg d\* in ((Q\#\text{WORDS}) \to (Q)) by \text{df intro} star\_intro

\gg Q in \text{STATES}

\gg d in ((Q\#\text{SYMBOLS}) \to Q)

H \gg w in L(M) by \text{df intro} accept\_equality

\gg w in \text{WORDS}

\gg d(q0,w) = qf in \text{Q} - \text{where } (Q,d,q0,qf) = M-

H \gg d\*(q,w) in T by \text{df intro} \text{delstar\_equality\_apply}

\gg T in \text{TYPE}

\gg d in (T\#\text{SYMBOLS}) \to T

\gg q in T

\gg w in \text{WORDS}
Ic: Elimination
-------------

H, M, DFA, H' >> T by dfelim M
  dfa_elim hyp
  Q: STATES, d: ((Q) # SYMBOLS) -> (Q), q0: (Q), F: P((Q)), M: = (Q, d, q0, F) in DFA
  >> T[(Q, d, q0, F) / M]

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

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

Id: Computation
-----------------

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

H >> d*(q, e) = q' in (Q) by dfreduce 1
  star_reduction_eps no
  >> a = q' in (Q)

H >> d*(q, v * s a) = q' in (Q) by dfreduce 1
  star_reduction_antics no
  >> d(d*(q, v), a) = q' in (Q)
  (>> v in WORDS
   >> a in SYMBOLS
   >> Q in STATES
   >> q in (Q)
   >> d in ((Q) # SYMBOLS) -> (Q))

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

II Tactics
---------

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

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

III-introduction
----------------

H >> Q: STATES, T
  Q: STATES >> T

H >> YM: DFA, T
  M: DFA >> T
  dfa_all_intro

H >> Yw: L(M), T
  w: L(M) >> T
  accept_all_intro

H >> YM: DFA, T
  M: DFA >> T
  YM: DFA, Q: STATES, d: ((Q) # SYMBOLS) -> (Q), q0: (Q), F: P((Q)), M: = (Q, d, q0, F) in DFA
  >> [(Q, d, q0, F) / M]

H >> Yw: L(M), T
  w: L(M) >> M in DFA
  w: L(M), v: WORDS, v e L(M), v = w in WORDS >> T[v / w]

Try to prove the unnecessary subgoals of star reduction, dfa equality apply
star_antics Compute no
star_concat Compute no
delstar apply