The Oyster
Proof Development System

Christian Horn

December 23, 2016

Abstract

This report describes a logical system called Oyster, based on intuitionistic type theory, and a prototype implementation [1] in the form of a logic programming system. Oyster is a synthesis of the ideas of constructive logic, Martin Löf type theory, Constable's refinement logic and logic programming. It is a rational reconstruction of the Nuprl (pronounced "nu-pearl") system built at Cornell. This document presents the system exactly in its internal form, i.e. it *is* the formatted source code itself. Although this implies some difficulties in the presentation of the material, it has the important advantage, that there is no difference between the logic documented and the logic implemented. In fact this document is not only a reference manual, but the precise and executable formal specification of a logical system: perhaps one step towards increasing the reliability of theorem proving and, more generally, of reasoning systems.

```
oyster_version('$Id: oyster.pl,v 1.33 2008/05/22 16:55:16 smaill Exp $').
info :- oyster_version(X),
        write('Oyster release '),
        write(X),nl.
```


[1] The prototype implementation is jointly distributed by the University of Edinburgh and the Humboldt-University at Berlin

Acknowledgements

This research was carried out during a visit of the author at the *Department of Artificial Intelligence* of the *University of Edinburgh*. This work was financially supported by a grant from *The British Council*. It was the influence of *Alan Bundy*, who led me to the ideas of intuitionistic type theory, and the fascinating ideas floating around in his *Mathematical Reasoning Group*, which encouraged me to do this work. Thanks to all those who gave their time to discuss the ideas, to read various drafts of this papers and to fix up a lot of bugs in the implementation, in particular to Frank van Harmelen, Alan Smaill and Andrew Stevens. Thanks go to my former home institution, the *Department of Mathematics* of the *Humboldt-University at Berlin*, in particular to those who have made this visit possible.

Contents

1 Introduction **1**
1.1 The Oyster System 1
1.2 The User Interface 2
1.3 Representing Theorems 2
1.4 Representing Proof Trees 6
1.5 Global Control Predicates 9
1.6 Focusing 12
1.7 Display Routines 14
1.8 Selector Predicates 15
1.9 The Inference Engine 18

2 The Rule Base **28**
2.1 Overview 28
2.2 Atom 32
2.3 Void 34
2.4 Pnat 35
2.5 Pless 38
2.6 Int 41
2.7 Less 44
2.8 List Types 48
2.9 Disjoint Union Types 50
2.10 Function Types 53
2.11 Dependent Function Types 56
2.12 Product Types 60
2.13 Dependent Product Types 62
2.14 Quotient Types 65
2.15 Subset Types 67
2.16 Recursive Types 70
2.17 Acc Types 73
2.18 Membership and Equality 75

2.19 Universes 77
2.20 Miscellaneous 77
2.21 Shorthands 82

3 Tactics **84**
3.1 Mark and Copy 84
3.2 Equality and Arithmetic 85
3.3 Syntax Checker for Type Theoretical Terms 86
3.4 Pretty Printer for Type Theoretic Terms 89
3.5 Substitutions 92
3.6 Free variables 95
3.7 Convertible Terms 97
3.8 Support for Peano Natural Numbers 97
3.9 Legality of Recursive Types 98
3.10 Type Checking 99
3.11 Rewrite Rules 100
3.12 Evaluation 101
3.13 Computation of tagged terms $[[t]]$ 104
3.14 Generate / Check tagging for compute with using() clause 106
3.15 Manipulations of the hypothesis list 109
3.16 Utilities 110

A Examples **114**
A.1 Factorial 114

Chapter 1

Introduction

1.1 The Oyster System

Oyster is an *Interactive Proof Editor* for goal directed backward chaining proofs. It is not an automatic theorem proving system. As a *programming environment* it combines a *structure editor* with a *verification condition generator*, but it is neither a program synthesiser nor an automatic program verification system. Oyster is a *framework* for developing application oriented intelligent systems, which integrate facilities for creating new theories, proving theorems, synthesising and running programs. Oyster provides the kernel functions needed for such systems and a meta language, which supports the development of higher level strategies for theorem proving as well as for program synthesis.

The main idea behind the Oyster development was that by using an appropriate *metalanguage* it should be possible to reformulate the current knowledge about theorem proving and program synthesis, in the hope of discovering more general principles. Oyster uses *Edinburgh Prolog* as a meta language, because it is a well-known, modern, expressive, and widely used language, which is especially suited for this purpose because of its unification and backtracking mechanism. Prolog is suitable as a command language as well as a language for implementing a specialised user interface. There already exist a large number of AI systems, written in Prolog, which could be used directly for applying heuristics and learning on the level of the meta theory.

Oyster provides a set of built-in interface predicates for basic operations on the proof tree including the application of a single inference step. First experiments with Oyster have shown that already these simple operations along with with the possibility of stringing them together into new commands using the Prolog mechanisms of unification and backtracking forms a very powerful command language. The extent of the full expressiveness of this command language is one open research topic. In

this document we have included only some small examples of the use of Prolog as a meta level language, to give the reader at least a feeling for the potentials of this approach. Appendix A summarises the built-in predicates, which support the use of Prolog as a meta-language.

Type theoretic terms are represented as Prolog expressions using the built in operator declarations of Prolog. This allows us to apply the Prolog unification mechanisms directly to type theoretic terms. There are special built-in predicates providing substitution and higher order unification on type theoretic terms.

In any stage of the proof development process it is possible to access the *extract term* of the proof constructed so far. Open subgoals of the proof, insofar as they have any constructive significance and are not only wellformedness goals, correspond to Prolog variables in the extract term.

There is a built-in evaluator for type theoretic terms, which allows the direct execution of Oyster programs. This evaluator is a two place Prolog predicate, which may be used as part of a control predicate. This opens the possibility of using the extract term of meta level theorems directly for controlling object level reasoning processes. With further development of the Oyster language it could become possible to replace Prolog as a control and meta language by Oyster itself. If we have reached this point, then we are near to the next generation of programming languages and systems.

1.2 The User Interface

The implementation of the Oyster system could be considered as an extension of Prolog by an *abstract data type*, providing some low level interface predicates for building more advanced programming and theorem proving environments. We use an operational specification for this abstract data type in terms of a "fictive" two layer Prolog implementation, which is not directly accessible to the user. The basic layer describes the internal representation of the proof tree and elementary operations on the proof tree (like positioning, assignment and extraction operations of subtrees). In section 2.2 we give a Prolog definition of these basic operations, but they should be considered as an implementation modell only. The second layer consists of the interface predicates described throughout this chapter and the rule base given in the next chapter. The interface predicates are given in terms of an operational specification using Prolog extended by the elementary operations of the first layer.

1.3 Representing Theorems

This section describes the representation of theorems and type theoretic terms in Prolog by means of operator declarations and summarises the implications of this

approach for the user. The main structure of the terms is identical with the representation in the Nuprl system, however there are some small differences mostly related to the use of Prolog as implementation language. For the interpretation of the terms, please refer to the next chapter (*Rule Base*). The aim of this section is only to give an overview and a summary of the syntax. The predicate $syntax(H, X)$ is used to check whether or not X is a type theoretic term under the assumption of the declarations and definitions in H.

- Theorems are terms of the form H ==> G, where H is a possibly empty (Prolog) list of hypotheses H_i and G is the goal to be proven. Hypotheses are *definitions*, references to other *theorems*, or *assumptions*.
- *Definitions* have the form $d(x, y, ...)$ <==> $t_{x,y,...}$, where d is an arbitrary identifier, $x, y, ...$ form a possibly empty parameter list and $t_{x,y,...}$ is any type theoretic term with free variables $x, y, ...$ Definitions can also be defined globally using the $create\_def$ predicate. The following example illustrate its use:

```
| ?- create_def( user ).
|: plus(x,y) <==> x+y.

yes
```

  Calling $save\_def(Name, Filename)$ will save the defined term $Name$: into the UNIX file $Filename$. The current defs can be determined by backtracking with $current\_def(X)$, and the def $Name$: erased with $erase\_def(Name)$.

- References to *theorems* have the form v : H==>G or v : H==>G *ext* E , where H and G are a list of hypothesis and the goal of the theorem again, and E is the extract term derived from that proof. The theorem name v is used only locally, but should reflect the common interpretation of the theorem. These theorem references have been introduced to document the relations between theorems. They depict the fact that the proof of the current theorem refers only to the theorems mentioned in the hypothesis list; and it may refer either only to the fact that these theorems are valid or it may refer to the proof construction of these theorems too. A library system on top of the current Oyster system has to prevent circular reasoning over theorems.
- *Assumptions* are of the form $v : T_i$ stating that the type T_i is inhabited by the element v, which may be read as:"v is declared to be of the type T_i" (and in so far it is guaranteed that T_i is inhabited at least by v) or "v is the name of an assumption represented by T_i" (and v is a symbolic name for a proof of T_i).

- The goal G is an arbitrary type, i.e. a type theoretic term belonging to some universe. The aim of the proof is to show, that G is inhabited, by explicitely constructing a member. This construction process might be characterized as stepwise refinement proof yielding the extract term E, which is guaranteed to be a member of G.

- Type theoretic variables are written as sequences of letters and digits and underscores, starting with a small letter. If necessary variables are modified by adding underscores. Completely new variables are generated from the sequence $v0, v1, v2, ...$. Although the system keeps track of the variables used so far, you should generally avoid these variables to prevent confusion.

- Universes are written in the form $u(i)$, where i is a positive integer. The basic types *atom, void, pnat* and *int* are written as they are.

- Atoms "..." are written as Prolog terms of the form $atom('...')$, to avoid ambiguities between type theoretic atoms and other constructs represented by Prolog atoms. The test of equality of atoms is a basic operation, therefore there is a decision operator for atoms in the form $atom\_eq(a, b, s, t)$.

- There is one impossibility operator of the form *any(x)* which maps a proof of a contradiction into any type.

- The type *pnat* represents the natural numbers with Peano arithmetic. The elements of *pnat* have the form 0 or $s(...)$, where $s(...)$ is the *successor* function on natural numbers. Inductive definition terms have the form $p\_ind(a, b, [x, y, t])$. The decision operators have the form $pnat\_eq(a, b, s, t)$, $pless(a, b, s, t)$.

- Integers are encoded as ordinary integer numbers in Prolog. The integer operations $-a$, $a + b$, $a - b$, $a * b$, a/b, and $a \ mod \ b$ as well as the ordering relation $a < b$ and $a <= b$ are written as usual. The decision operators have the form $int\_eq(a, b, s, t)$, $less(a, b, s, t)$. Inductive definition terms have the form $ind(a, [x, y, s], b, [u, v, t])$.

- List types have the form A *list*, where A is the base type. The empty list has the form *nil* and the usual list construction operation is written in the form $a :: b$, with a being an element of A and b being an element of the list type. List induction terms have the form $list\_ind(a, s, [x, y, z, t])$.

- Disjoint union types are written in the form $A\backslash B$, where A and B are arbitrary types. The injection operators have the form *inl(a)* and *inr(b)*, with a and b being terms of the type A or B respectively. The general decision operator has the form $decide(u, [x, s], [y, t])$.

- The product types have the form $A\#B$ or $x{:}A\#B$, with A and B being types and x being a free variable in B, which becomes bound in the product type. Their elements are pairs of the form $a\&b$, where a and b are arbitrary terms of type A and B respectively, and the generalised projection operator is written in the form *spread(a,[x,y,t])*.

- The function types $A \rightarrow B$ and $x : A \rightarrow B$ have the form $A => B$ and $x : A => B$; λ-terms have to be written in the form $lambda(x, b)$, where b is an arbitrary term of the type B and x is a free variable in B which is assumed to vary over A. x becomes bound in the λ-term. Function application terms have the form $f\ of\ a$, where f is an arbitrary term (member) of a function type and a is a term of the domain type of that function type.

- Quotient types have the form $A///[x, y, E]$, where A is the basic type and E is an equivalence relation in the free variables x and y, which become bound in the quotient type. The equivalence classes are written in the form $\{x\}$ for arbitrary x in A.

- Set types are written in the form $\{x : A\backslash B\}$, where A is the base type and B is a type with the free variable x defining the characteristic property of the set. x becomes bound in the set term.

- Recursive types have the form $rec(z, A\backslash B)$ where A and B are types and z is a free variable in B, which does not appear in A. Type induction terms are written in the form $rec\_ind(r, [h, i, t])$.

- Equalities and membership relations have the form $a{=}b\ in\ t$ and $a\ in\ t$, respectively. Multiterm equalities are not allowed.

- The *extract term* of a theorem, reflecting the algorithmic ideas behind the proof of that theorem, is written in the form $term\_of(t)$ where t is the name of a theorem referred to. All theorems used have to be declared in the hypothesis list of the top level goal. If you make proper use of the form of the extract term (for example by rewriting or evaluating the extract term), ensure that the reference in the hypothesis list really contains the extract term.

The priority and associativity of the operators is best described by their Prolog declarations.

```
?-
  op(950,xfx,[ext]),
  op(900,fx, [==>]),
  op(900,xfx,[==>, <==>]),
  op(850,xfy,[<=>]),
```

```
op(850,xfy,[:, =>, #, \, ///]),
op(700,yfx,[in, =, <, <*]),
op(500,yfx,[+, -]),
op(500,fx, [-]),
op(400,yfx,[*, /, mod]),
op(300,xfy,[&, ::]),
op(250,yfx,[of]),
op(200,xf, [list]).
```

1.4 Representing Proof Trees

The Oyster systems works internally with an explicit representation of (partial) proof trees in form of Prolog terms $\pi(H ==> G, R, E, S)$ which are for each theorem t assigned to the global variable $\vartheta_{theorem}(t)$.[1] $H ==> G$ is the theorem to be proved,

[1] The use of global variables, although not typical for Prolog, is essential for this implementation. Thats why we have introduced the shorthand notations ':=' for assignment and '=:' for dereferencing of variable values. As a result of this global storage strategy, it sometimes happens that the database gets cluttered with intermediate datastructures. The predicate *cleandatabase* removes all these intermediate datastructures. Obviously, this predicate should **never** be called inside a proof, but only at top-level, to repair damage after things have gone wrong

```
?-op(900,xfx,[':=','=:']).

?-dynamic dynvalue/2.
```

Autoloading of unknown syntax via the Clam library mechanism

```
?-dynamic autoload_defs_ass/1, autoload_defs_ass2/1, autoloading_def/1.
autoload_defs_ass(no).
autoload_defs(yes) :-
    retractall(autoload_defs_ass(_)),
    assert(autoload_defs_ass(yes)).
autoload_defs(no) :-
    retractall(autoload_defs_ass(_)),
    assert(autoload_defs_ass(no)).
autoload_defs(_) :-
    write('Please use autoload_defs(yes) or autoload_defs(no)'),nl,
    write('Currently, autoloading is set to '),
    autoload_defs(S),
    write(S),nl.
```

```
ϑ_theorem(N) := _ :-
    recorded(N,ϑ_theorem(N,_),R),erase(R),
    recorded(ctheorem,N,RR),erase(RR),
    fail.
ϑ_theorem(N) := V :-
```

```
      (var(V); recorda(N,ϑ_theorem(N,V),_),recorda(ctheorem,N,_)),!.

ϑ_pos(N) := _ :-
      recorded(N,ϑ_pos(N,_),R),erase(R),fail.
ϑ_pos(N) := V :-
      (var(V); recorda(N,ϑ_pos(N,V),_)),!.

cdef(N) := _ :-
      recorded(N,cdef(N,_),R),erase(R),
      recorded(cdef,N,RR), erase(RR),
      fail.
cdef(N) := V :-
      (var(V); recorda(N,cdef(N,V),_),recorda(cdef,N,_)),!.

V:=_ :- Θ'_(ϑ(V,_)), fail.
V:=X :- (var(X); Θ_+(ϑ(V,X))),!.

ϑ_theorem(N) =: X :-
      !,
      recorded(ctheorem,N,_),
      recorded(N,ϑ_theorem(N,X),_).
ϑ_pos(N) =: X :-
      !,
      recorded(N,ϑ_pos(N,X),_).
cdef(N) =: X :-
      !,
      recorded(cdef,N,_),
      recorded(N,cdef(N,X),_).
V=:X :- Θ_?(ϑ(V,XX)),!,X=XX.



cleandatabase :-
      recorded(ctheorem,T,R), functor(T,_,N), N>=1,
      erase(R),
      recorded(T,ϑ_theorem(T,_),R1), erase(R1),
      recorded(T,ϑ_pos(T,_),R2), erase(R2),
      Θ_(ϑ(ϑ_thm,T)),
      fail.
cleandatabase.



Θ_+(ϑ(X,Y)) :- recorda(X, ϑ(X,Y), _).
Θ_?(ϑ(X,Y)) :- recorded(X, ϑ(X,Y), _).
Θ_(ϑ(X,Y)) :- recorded(X, ϑ(X,Y), R), erase(R).



Θ'_(X):-Θ_(X),fail.
Θ'_(_).
```

R is the refinement applied to the top level, E the resulting extract term and S the list of subproblems generated by that refinement. If a problem is still open, then R, E and S are unbound Prolog variables. The list of subproblems again consists of $\pi(H_i ==> G_i, R_i, E_i, S_i)$ terms, each describing a subproblem. The hypothesis lists of subproblems contain only the increment to the hypotheses of the surrounding problem. In that sense the logic behaves strictly monotonically: The complete hypothesis list for a certain node in the proof is computed during the process of accessing that node. The extract term is stored only to the extend corresponding to the one refinement step carried out at that level. This extract term may contain Prolog variables, which correspond to the extract terms of some of the subproblems. Those subproblems which have an influence on the extract term of the surrouding problem are stored in the form $\pi(H_i ==> G_i, R_i, E_i, S_i)$ ext V_i where V_i is one of the Prolog variables appearing in the extract term E of the surrounding problem. This means that the extract term corresponding to a whole subtree is computed only on request simply by recursively binding all variables V_i to the corresponding E_i

```
△=:([],P,P).
△=:( [1|L], π(G,universe(I),E,S),S0):-
    ϑ_universe:=I,
    △=:( [1|L],π(G,~,E,S),S0).
△=:( [1|L],π(G,autotactic(T),E,S),S0) :-
    ϑ_autotactic:=T,
    △=:([1|L],π(G,~,E,S),S0).
△=:( [N|L],π(_,_,_,S),π(HH==>G,R,E,Sx)):-
    !,
    △=:(N,S,SS),
    △=:(L,SS,π(HH==>G,R,E,Sx)).

△=:(N,L,_) :- integer(N),var(L), !, fail.
△=:(1,[S ext _|_],S):-!.
△=:(1,[S|_],S):-!.
△=:(N,[_|L],X):-integer(N),N>1,N1 is N-1,!,△=:(N1,L,X).


△:=([],π(H==>G,_,_,_),π(_==>G,R,E,S),π(H==>G,R,E,S)).

△:=([P|L],π(G,R,E,S),N,π(G,R,E,T)):-
    △=:(P,S,SS), △:=(L,SS,N,NN), △:=(P,S,NN,T).

△:=(1,[_ ext E|T],N,[N ext E|T]):-!.
△:=(1,[_|T],N,[N|T]).
```

$\triangle_{:=}$(I,[H|T],N,[H|S]):-integer(I),I>1,J is I-1,$\triangle_{:=}$(J,T,N,S).

extract(π(_,_,_,S),_):-var(S),!.
extract(π(_,_,E,S),T):-*extract*(E,S,T).

extract(E,[],E).
extract(E,[P *ext* E0|S],EE):-*extract*(P,E0),!,*extract*(E,S,EE).
extract(E,[_|S],EE):-*extract*(E,S,EE).

1.5 Global Control Predicates

There exists no special library system as a part of Oyster. The main idea of the system design was to supply all the basic operations which allow the independent implementation of possible different user interfaces on top of Oyster. Oyster assumes that theorems are stored in separate files. The user can load and save theorems independently. Describing a problem means creating a file of the form "$H ==> G$.", where H is a hypothesis list and G is the goal to be proven (don't forget the dot at the end! It's a Prolog term). Creating a theorem means loading this problem description with the $create\_thm(thm, file)$ predicate. Theorems can be created on-line using $create\_thm(thm, user)$. Once a theorem is created it can be saved and reloaded between the proof steps just as you want. The *create_thm* and *load_thm* functions perform a rough structure check to avoid errors which result from loading totally wrong files. Loading a theorem does not require reproving it. Therefore it seems to be adequate to use the save and reload mechanism for version handling and runtime memory reorganisation. Tactics and your own proof development commands should be saved in project oriented Prolog source files and loaded using the common $consult(file)$ mechanism. If you are in any doubt whether your proof is still valid, for example after some confusion in your file system, you can force the system to reprove it. At least at the very end of a proof you should check it seriously.

```
create_thm(T,F):-
    readfile(F,H==>G),
    add_thm( T, H==>G ).


add_thm( T, H==>G) :-
    retractall(autoloading_def(_)),
    (autoload_defs_ass(yes) ->
    assert(autoload_defs_ass2(yes));
```

```
    retractall(autoload_defs_ass2(_))),
    syntax(H==>G),
    ϑ_theorem(T):=π(H==>G,_,_,_),
    ϑ_pos(T):=[].


load_thm(T,F):-
    load_thm(T,F,_).
load_thm(T,F,Q):-
    readfile(F,π(H==>G,R,E,S)),
    retractall(autoloading_def(_)),
    (autoload_defs_ass(yes) ->
    assert(autoload_defs_ass2(yes));
    retractall(autoload_defs_ass2(_))),
    syntax(H==>G),
    ϑ_theorem(T):=π(H==>G,R,E,S),ϑ_pos(T):=[],
    status_0(π(H==>G,R,E,S),Q).


save_thm(T,F):-
    ϑ_theorem(T)=:P,atom(F),tell(F),writep(0,P),write('.'),nl,told.


create_def( F ) :-
    readfile(F,Head<==>Body),
    add_def( Head <==> Body ).
```

Not just speed. the expansion of definitions does not always respect bindings. for example, $y <==> atom(y)$ will cause $y : t => \ldots$ to be unfolded to $atom(y) : t => \ldots$ etc.

```
add_def( {Name} <==> Body ) :-
    τ_var(Name),
    !,
    syntax( [], Body ),
    cdef(Name) := ({Name} <==> Body).
add_def( Head <==> Body ) :-
    Head =.. [Name|Args],
    τ_var(Name),
    check_set( Args ), % check for linearity of args
    (bagof( VT, Arg^(member(Arg,Args), VT = (Arg:dummy)), VarsHyps ) ->true;
    VarsHyps = []),
    syntax( VarsHyps, Body ),
    cdef(Name) := (Head <==> Body).
```

```
check_set([]).
check_set([H|T]) :- \+ member(H,T), check_set(T).


save_def(T,F) :-
     cdef(T)=:P,atom(F),tell(F),writeq(P),write('.'),nl,told.


current_def(X) :- cdef(X) =: _.


erase_def(D) :- cdef(D) := _.


erase_thm(T) :- ϑ_theorem(T) := _.



writep(_,π(H==>G,R,_,_)):-
     var(R),write('π('),writeq(H==>G),write(',_,_,_)').
writep(N,π(H==>G,R,E,S)):-
     \+ var(R), write('π('),writeq(H==>G),write(','),nl,
     tab(N),writeq(R),write(','),writeq(E),write(','),nl,
     tab(N),write('['),NN is N+1,writep(NN,S),nl,
     tab(N),write('])').
writep(_,[]).
writep(N,[P ext E]):-!,writep(N,P),write(' ext '),writeq(E).
writep(N,[P]):-!,writep(N,P).
writep(N,[P ext E|T]):-
     writep(N,P),write(' ext '),writeq(E),write(','),nl,tab(N),writep(N,T).
writep(N,[P|T]):-writep(N,P),write(','),nl,tab(N),writep(N,T).

writel([]) :- write('[]').
writel([H]) :- !,write('['),write(H),write(']').
writel([H|T]) :- write('['),write(H),writel1(T),write(']').
writel1([]).
writel1([H|T]) :- write(','),nl,write(H),writel1(T).
```

The default autotactic is bound to the token "defaultautotactic". The default for the default is *idtac*, and can be changed by assigning to "defaultautotactic" with :=.

```
?- defaultautotactic := idtac.
```

1.6 Focusing

The first step after loading or creating some theorems is to select a theorem for further work. Selecting a theorem sets the current working position (the *focus*) on that point in the proof of the theorem where it was when the theorem was left last time. That means, you can switch between theorems without losing the position information. Every theorem has its own local focus, which is the current focus if the theorem is selected. If a theorem is created or loaded, the local focus is set at the top of that theorem. There are predicates for absolutely and relatively positioning the current focus (and therefore the local focus of the theorem selected). If the focus moves around the proof tree, the value of global variables which correspond to properties of the proof tree (like the current autotactic, or the current universe universe level) are changed automatically according to the current position in the proof tree.

- *select*(*thm*) selects a theorem for further work and sets the current focus to local focus of that theorem. If you have forgotten to select a theorem most of the predicates will fail. *select*(*X*) with a variable parameter gives you first the name of the theorem currently selected (if any), and then (using the ordinary Prolog backtracking mechanism) the names of all loaded theorems; *select* without any parameter displays the name of the currently selected theorem.

```
select(X):-var(X),ϑthm=:T,!,(X=T;(ϑtheorem(X) =: _,\+X=T)).
select(X):-var(X),!,ϑtheorem(X) =: _.
select(N):-
    ϑtheorem(N)=:_, ϑthm:=N, ϑpos(N)=:L,pos(L),ϑstatus:=nonpure,
    (functor(N,Name,Arity),
    SA is Arity+1,
    functor(NN,Name,SA),
    ϑtheorem(NN) := _,
    ϑpos(NN) := _ ),!.
select:-
    ϑthm=:X,!,
    (X=..[XX,N],integer(N),select(XX),write(XX) ; write(X)),
    tab(1).
```

- *pos*(*x*) can be used in two ways: If x is not a fully instantiated list of integers, then it tries to match x with the current position. If x is a list of integers *pos*(*x*) switches the current focus to that position. Positions are specified as tree coordinates,i.e. as a list of integers which describes the top-down way to the node in terms of the number of the sons to which one has to branch on that way. *pos* without any parameter displays the current position.

```
pos(P):-var(P),ϑ_thm=:N,ϑ_pos(N)=:P,!.
pos(P):-
    ϑ_thm=:N, ϑ_theorem(N)=:X, ϑ_universe:=1,
    defaultautotactic =: AT, ϑ_autotactic:=AT,
    △_=:(P,X,Y), ϑ_pos(N):=P, ϑ_π:=Y,!.
pos:-pos(P),!,write(P),tab(1).
```

- *top* resets the focus to the top of the current theorem;

```
top:-pos([]),!.
```

- *up* moves the focus up one level, in the case of backtracking it tries to move further up;

```
up:-pos(P),append(Q,[_],P),(pos(Q);up).
```

- *down*(i) focuses down to the i-th subproblem, if there is one, and returns the focus in the case of backtracking to the starting position. If it is not possible to go down to the i-th subproblem, *down*(i) fails. *down*(X) may be used for generating the direct subgoals one after the other. *down*(X) is a backtrackable predicate which resets the focus to the starting point. *down* (without any parameter) is a shorthand notation for *down*(_).

```
down(X):-
    integer(X),pos(P),( append(P,[X],Q),pos(Q);(pos(P),!,fail) ).
down(X):-
    var(X),pos(P),ϑ_π=:π(_,_,_,S), \+ var(S),
    (generate_sons(S,1,X),append(P,[X],Q),pos(Q);pos(P),!,fail).
down:-down(_).
```

- *next*(X) sets the focus on the next unsolved subproblem in the subproof below the current position, and generates in the case of backtracking successively all other unsolved subproblems, until it resets the focus to the starting point and fails. The parameter is always bound to the position of the open problem just found. *next* (without any parameter) is a shorthand notation for *next*(_).

```
next(Q):-
    pos(P),P=Q,ϑ_π=:π(_,R,_,_),var(R),!.
next(QQ):-
    pos(P), ϑ_π=:π(_,_,_,S),
```

```
    ( generate_sons(S,1,I), append(P,[I],Q), pos(Q), next(QQ) ; pos(P), fail ) .
next:-next(_).
next_goal(G) :-
    next_goal(_,G,_).
next_goal(Q,G,Rule):-
    pos(P),P=Q,ϑπ=:π(G,Rule,_,_).
next_goal(QQ,G,Rule):-
    pos(P), ϑπ=:π(_,_,_,S),
    ( generate_sons(S,1,I), append(P,[I],Q), pos(Q),
      next_goal(QQ,G,Rule) ; pos(P), fail ) .
next_⊢(Q,Rule):-
    pos(P),P=Q,ϑπ=:π(_,Rule,_,_).
next_⊢(QQ,Rule):-
    pos(P), ϑπ=:π(_,_,_,S),
    ( generate_sons(S,1,I), append(P,[I],Q), pos(Q),
      next_⊢(QQ,Rule) ; pos(P), fail ) .


generate_sons([_|_],I,I).
generate_sons([_|T],N,I):-N1 is N+1,generate_sons(T,N1,I).
```

1.7 Display Routines

There are two display predicates *display* and *snapshot*. *display* is the normal way to show the current problem. $snapshot(filename, depth)$ writes an overview of the proof tree below the current focus on the file specified, and is mostly used for documentation. The second parameter restricts the depth to which the proof tree is displayed, 0 implies no restriction, i.e. the documentation of the complete proof tree. There are three global parameters which allow some tuning of the layout yielded by *display* and *snapshot*. The *screensize* parameter should be equal to the current width of the window (standard value is 80), the $shift$ parameter describes the indentation the system automatically chooses for each new level of declaration or operators if they don't fit on the rest of the line (standard value is 2), and the $fringe$ parameter gives the nesting depth, below which the fringed output would yield only '...'. the $fringe$ standard value 0 has no effect and keeps all terms unchanged. The parameters may be checked and changed at any time using the corresponding $screensize(X)$, $shift(X)$ or $fringe(X)$ predicate.

```
screensize(X):-(integer(X),0<X,ϑscreensize:=X,!; ϑscreensize=:X).


shift(X):-(integer(X),0=<X,ϑshift:=X,!; ϑshift=:X).
```

```
fringe(X):-(integer(X),0=<X,ϑ_fringe:=X,!; ϑ_fringe=:X).

?- screensize(80), shift(2), fringe(0).



display:-snapshot(user,2).


snapshot:-snapshot(user,0).
snapshot(F):-snapshot(F,0).
snapshot(F,D):-
    atom(F),integer(D),tell(F),ϑ_thm =: CT,write(CT),write(': '),pos,status,
    (universe(1); universe),(autotactic(repeat(intro)); autotactic),nl,
    ϑ_π=:P,snapshot(D,0,1,P),told,!.

snapshot(D,Z,N,π(H==>G,R,_,S)):-
  ϑ_shift=:C,ZZ is Z+C, write_hyp(Z,N,NN,H),
  tab(Z),write('==> '),writeterm0(ZZ,G), DD is D-1,
  (\+ DD=0,tab(Z),write('by '),writeterm0(ZZ,R),nl,snapshot(DD,ZZ,1,NN,S); nl).

snapshot(_,_,_,_,S):-var(S),!.
snapshot(_,_,_,_,[]).
snapshot(D,Z,I,N,[PP|S]):-
  (PP=(P ext _); PP=P),tab(Z),write([I]),status_0(P,Q),tab(1),write(Q),nl,
  snapshot(D,Z,N,P),J is I+1,snapshot(D,Z,J,N,S).



write_hyp(Z,StartHyp, LastHyp, Hyps ) :-
    write_hyp(Z, 1, StartHyp, LastHyp, Hyps ).

write_hyp(_, Ctr, _, Ctr, [] ).
write_hyp(Z, Ctr, StartHyp, LastHyp, [_|R] ) :-
    Ctr < StartHyp,
    !,
    NCtr is Ctr + 1,
    write_hyp( Z, NCtr, StartHyp, LastHyp, R ).
write_hyp(Z, Ctr, StartHyp, LastHyp, [HH|R] ) :-
    tab(Z),write(Ctr),write('. '),ZZ is Z+3,writeterm0(ZZ,HH),
    NCtr is Ctr + 1,
    write_hyp(Z,NCtr,StartHyp,LastHyp,R).
```

1.8 Selector Predicates

The selector predicates operate on the current focus without modifying the proof tree.

- The *status*(X) predicate is used to check the current status. It yields one of the values *complete*, *complete*(*because*), *partial*, or *incomplete*; a proof marked *complete*(*because*) contains some uses of the *because* inference rule, but is otherwise complete.
- The *goal*(G) predicate can be used to check the structure of the current goal;
- The *hypothesis*(X) predicate generates all hypotheses matching the given pattern term X and fails if there are no (further) hypotheses;
- The *hyp_list*(X) predicate yields the list of all hypotheses;
- The $refinement(X)$ predicate allows to pick up the complete (i.e. not fringed) current refinement, which is useful for writing transformational tactics; and
- *extract*(X) yields the 'polished' extract term. Polishing means here that all bound variables appearing in the extract term are checked, to see whether they are really used or not. If not their defining occurence is replaced by ~;
- *autotactic*(X) and *universe*(X) allow a check of the current autotactic and universe level, respectively. The current values of autotactic and universe level are computed and stored in the process of top-down positioning in the proof tree;
- the short forms *select, pos, status, goal, ...* without any parameters may be used for direct display of the current values, which is especially useful when working on small or dumb displays.

The combination of selector predicates with focusing predicates gives a powerful control mechanism, which is demonstrated by the following examples of user defined control predicates:

```
forall(G,X):-
   top,next,copy(G,GG),goal(GG),apply(X),fail.
upwards:-up,status(partial).
```

$forall(G, X)$ applies the tactic X to all open subgoals of the structure G; and *upwards* moves the focus upwards to the first level which is not completely solved.

status(Q):-ϑ_π=:P,$status_0$(P,Q),!.

goal(G):-ϑ_π=:π(_==>G,_,_,_),!.

hyp_list(H):-ϑ_π=:π(H==>_,_,_,_),!.

hypothesis(H):-hyp_list(L),!, member(H,L).

refinement(R):-ϑ_π=:π(_,R,_,_),!.

extract(T):-ϑ_π=:P,$extract$(P,T0),polish(T0,TT),!,T=TT.

autotactic(X):-$\vartheta_{autotactic}$=:X,!.

universe(X):-$\vartheta_{universe}$=:X,!.

```
status:-status(Q),write(Q),tab(1).
goal:-goal(G),write('==> '),writeterm(3,G).
hyp_list:-hyp_list(H),write_hyp(0,1,_,H).
refinement:-refinement(R),writeterm(R).
extract:-extract(T),writeterm(T).
autotactic:-autotactic(X),write(autotactic(X)),tab(1).
universe:-universe(X),write(universe(X)),tab(1).
```

```
status0(P,S) :- status1(P,SS),!,S=SS.
status1(π(_,_,_,S),incomplete):-var(S),!.
status1(π(_,R,_,_),P):- R==because,!,P=complete(because).
status1(π(_,_,_,S),P):- \+ var(S),status1(S,P).
status1(P ext _,S):-status1(P,S).
status1([],complete).
status1([H|T],S) :- status1(H,S1),status1(T,S2),
                    combine_status(S1,S2,S),!.
combine_status(X,X,X).
combine_status(incomplete,_,partial).
combine_status(partial,_,partial).
combine_status(complete(because),complete,complete(because)).
combine_status(complete(because),_,partial).
combine_status(complete,complete(because),complete(because)).
combine_status(complete,_,partial).
```

```
polish(X,Y):-var(X),!,X=Y.
polish([T],[TT]):-!,polish(T,TT),!.
polish([U|T],[U|T]) :- member(V,[U|T]),var(V),!.
polish([U|T],[~|TT]):- \+ appears(U,T),!,polish(T,TT).
polish([U|T],[U|TT]):-!,polish(T,TT).
polish(σ(T,Y,X),TT):-s(T,Y,X,T0),!,polish(T0,TT).
polish(lambda(U,T),lambda(UU,TT)):-!,polish([U,T],[UU,TT]).
polish(U:T,UU:TT):-!,polish([U,T],[UU,TT]).
polish(T,TT):-T=..[F|A],polish'(A,AA),TT=..[F|AA].


polish'([],[]).
polish'([X|T],[XX|TT]):-polish(X,XX),polish'(T,TT).
```

1.9 The Inference Engine

One of the main problems in organising the theorem proving process is to ensure the consistency of the data structures representing the current state of the proof. In the Oyster implementation, the proof tree itself is completely hidden from the user. The user can move around the focus and display parts of the proof tree, but for modifying the proof tree there is only one predicate (*apply*), which invokes the inference engine of the Oyster system, i.e. the only way of modifying the proof tree is the successful application of some sequence of inference rules.

There are two modes of operation of the inference engine: the *pure* and the *nonpure* (i.e.standard) mode. In the *pure* mode the autotactic as well as all the derived rules are temporarely switched off. This mode is useful in debugging proof trees and for the application of user defined meta-level strategies. The *nonpure* mode is the standard one and makes the system together with the standard autotactic (*repeat intro*) more suitable for creative interactive use.

1.9.1 The *apply*-Predicate

The *apply* predicate is parametrised by tactics or pseudo tactics. A tactic is either the specification of a inference rule, like *intro* or $elim(x)$, a call of a user defined tactic in the form $tactic(U, X)$ (or U for short), or an arbitrary combination of tactics by means of the predefined tacticals *then*, *or*, *repeat*, *complete*, and *try*. The semantics of the *tacticals* is described by the $\models(...)$ predicate. After the execution of the arguments of *apply* the current autotactic is *try*-applied to all generated subgoals (see below for the definition of the *try* tactical). The autotactic is considered as a property of the proof tree and may differ in several parts of the proof tree. The autotactic is assigned as an attribute to the proof tree and selected during the

process of top down positioning in the proof tree. Invoking the *autotactic* as well as the execution of *pseudo tactics* is described by the $\models_0(...)$ predicate.

The apply predicate modifies the proof tree only in the case of successful tactic application. Inference rules and tactics are applied by the backtrackable *prove* predicate. $apply(X)$ may be useful to obtain some hints about rules which are applicable in this situation. Some shorthand notations allow more convenient use.

```
apply(X):-
    var(X),!,
    toggle(direction,[backwards,forwards]),
    ϑπ=:π(G,_,_,_),(⊢(X,G ext _,_);⊢(X,G,_)),
    toggle(direction,[forwards,backwards]).
apply(R):-
    direction:=forwards, ϑπ=:P, !, ⊨0(R,P,Q), ϑπ:=Q,
    ϑthm=:T, ϑpos(T)=:H, ϑtheorem(T)=:M, △:=(H,M,Q,N), ϑtheorem(T):=N,!.

apply(R,H==>G,S):-
    ⊨(R,π(H==>G,_,_,_),π(_,_,_,Q)),unfold(Q,S).


unfold([],[]).
unfold([π(P,_,_,_) ext _|T],[P|TT]):-unfold(T,TT).
unfold([π(P,_,_,_)|T],[P|TT]):-unfold(T,TT).
```

The $\models_0(R,...)$ predicate handles pseudo tactics and invokes the autotactic by expanding R to $R\ then\ ....$ Pseudo tactics are $universe(I)$, $autotactic(T)$, $pure(T)$, and *undo*. They may not be combined by means of tacticals, **although the system does NOT enforce this restriction**.

- The current universe level is a property of the proof tree. It is assigned to a node of the proof tree and defines the default value for the universe level in the subtree below that point. The default value is computed in a top-down manner during positioning in the proof tree. The top level is assumed to be labelled with the universe level 1. The pseudo tactic $universe(I)$ assigns this value to the proof tree.

  ```
  ⊨0(universe(I),π(H==>G,_,_,_),
          π(H==>G,universe(I),X,[π(H==>G,_,_,_) ext X])).
  ```

- Autotactics are considered as default reasoning strategies and assigned to the proof tree in the form of a pseudo tactic $autotactic(T)$. The autotactic is

valid for the subtree below the point, where it is set, except subtrees with another explicitly defined autotactic. The top level is assumed to be labeled with *repeat intro*.

$\models_0$(autotactic(T),π(H==>G,_,_,_),
π(H==>G,autotactic(T),X,[π(H==>G,_,_,_) *ext* X])).

- *undo* is the destructive pseudo tactic, which deletes previous results in proving the current problem, i.e. the *status* of the current node becomes *incomplete*. All the subproof below the current position is deleted.

$\models_0$(undo,π(G,_,_,_),π(G,_,_,_)).

- In all other cases it is assumed that $\models_0$ is parametrised by a tactic and the argument is passed to $\models$, possibly extended by the application of the autotactic (in *nonpure mode* only). The $\models$ predicate always operates with full hypothesis lists, therefore the resulting subproblems have to be cleaned up at the end. $increment(P, Q, Q')$ deletes all redundant hypotheses from the subgoals of Q and produces Q'. The pseudo tactic $pure(T)$ switches the system temporarily, i.e. for the execution of the tactic T in the, into the *pure mode*.

$\models_0$(pure(R),P,π(G,pure(R),E,S)):-
ϑ_{status}=:T,!,
(ϑ_{status}:=pure,$\models$(R,P,Q),increment(Q,π(G,R,E,S)),ϑ_{status}:=T;
ϑ_{status}:=T,fail).

$\models_0$(R,P,QQ):-
ϑ_{status}=:pure,!,
$\models$(R,P,Q),increment(Q,QQ).
$\models_0$(R,P,QQ):-
$\vartheta_{autotactic}$ =: idtac,!,
$\models$(R,P,Q),increment(Q,QQ).
$\models_0$(R,P,π(G,R,E,S)):-
$\vartheta_{autotactic}$=:X,
$\vartheta_{autotactic}$:= idtac,
($\models$(then(R,try(X)),P,Q), !; $\vartheta_{autotactic}$:= X, !, fail),
$\vartheta_{autotactic}$:= X,
increment(Q,π(G,_,E,S)).

```
increment(π(H==>G,R,E,S),π(H==>G,R,E,S)).

increment(_,[],[]).
increment(H0,[π(H==>G,R,E,S)|T],[π(HH==>G,R,E,S)|TT]):-
   diff(H,H0,HH),increment(H0,T,TT).
increment(H0,[π(H==>G,R,E,S) ext V|T],[π(HH==>G,R,E,S) ext V|TT]):-
   diff(H,H0,HH),increment(H0,T,TT).
```

1.9.2 The ⊨-Predicate

The ⊨(R,...) predicate is the kernel of the inference engine. It describes the recursive control of the inference process by means of the tacticals *complete, try, repeat, then,* and *or*, it recognises rule specifications and calls the ⊢(...) predicate, and it handles user defined tactics. ⊨(...) is invoked with full problem descriptions, i.e. full hypothesis lists. The reduction process is performed at he end.

- The syntax of the tacticals is best described by their operator declarations:

```
?-
   op(950,xfy,['then']),
   op(900,xfy,['or']),
   op(850,fx,['repeat','complete','try']).
```

- *idtac* is the identical tactic which has no effect at all. It is useful in combination with the *then* $[T_1, ...]$ tactical, giving the possibility of applying other tactics only to certain branches of the proof tree. It should be used as the auto tactic if you want to switch the auto tactic off.

```
⊨(idtac,π(G,_,_,_),π(G,idtac,E,[π(G,_,_,_) ext E])):-!.
```

- T_1 *then* T_2 is a tactic defined by applying first T_1 and then T_2 on all subgoals. T_1 *then* $[T_2^1, T_2^2, ...]$ is a tactic defined by applying T_1 first and then each of tactics T_2^i to the corresponding subgoals which have been generated by T_1, i.e. T_2^1 to the first subgoal, T_2^2 to the second subgoal etc.

```
⊨(T1 then T2,π(H==>G,_,_,_),π(H==>G,T1 then T2,EE,S)):-
     ⊨(T1,π(H==>G,_,_,_),π(H==>G,_,E,U)), \+ var(U),⊨'(T2,E,U,EE,S),!.
```

- T_1 *or* T_2 is a tactic which first applies T_1 and in the case of failure, caused by the application of T_1, makes an attempt to apply T_2.

```
⊨(T1 or T2,π(G,_,_,_),π(G,T1 or T2,E,S)):- ⊨(T1,π(G,_,_,_),π(G,_,E,S)).
⊨(T1 or T2,π(G,_,_,_),π(G,T1 or T2,E,S)):- ⊨(T2,π(G,_,_,_),π(G,_,E,S)),!.
```

- *repeat* T is an unconditional tactical which tries to apply T on the given problem and recursively *repeat* T on all subproblems generated by T. You could define *repeat* T as equivalent to (T *then repeat* T) *or idtac*. It generates as a result the list of all subproblems which can not be handled further by T, i.e. T would fail when applied once more to any of these subgoals. *repeat* combined with *intro*-rule is a very powerful tool, thats why the standard autotactic is set to *repeat intro*. In fact, the shorthand definitions for *intro* are carefully tuned to reach this high level performance. Therefore you should define your own autotactic following the pattern *repeat* (... *or intro or* ...).

```
⊨(repeat T,π(H==>G,_,_,_),π(H==>G,repeat T,EE,S)):-
    copy(T,T1),copy(T,T2),
    ⊨(T1,π(H==>G,_,_,_),π(_,_,E,U)),!,⊨′(repeat(T2),E,U,EE,S).
⊨(repeat T,π(G,_,_,_),π(G,repeat T,E,[π(G,_,_,_) ext E])):-!.
```

- *complete* T is a conditional tactical which is successful with the effect of T, if the application of T solves the problem completely, i.e. if T generates no new subgoals, and fails without any effect otherwise.

```
⊨(complete T,π(G,_,_,_),π(G,complete T,E,[])):-
    ⊨(T,π(G,_,_,_),π(G,_,E,Subs)),nonvar(Subs),Subs=[],!.
```

- *try* is a failure catching tactical. *try* T is successful every time. It has the effect of T if T is successful, and has no effect otherwise.

```
⊨(try T,π(G,_,_,_),π(G,try T,E,S)):- ⊨(T,π(G,_,_,_),π(G,_,E,S)),!.
⊨(try T,π(G,_,_,_),π(G,try T,E,[π(G,_,_,_) ext E])):-!.
```

- The selection of the appropriate inference rule is driven by pattern matching between the current goal and the rule specification on one hand and the set of inference rules on the other hand. The rule base itself is documented in the next chapter. The rules are organised in such a way that the rule parameters $new[v, ...]$, and $at(i)$ may be omitted. The specification of a rule may be simplified using Prolog variables, which become instantiated from the preceding context, or by means of anonymous variables, which is especially valuable in

the case of the *subst* and *compute* rules. This approach allows the reusability of rules in a different context. Rules are always called on a copy of the rule specification. Therefore variable bindings may only be used inside a single rule specification to formalise structural restriction on the parameters, and so you will never get the values assigned to the variables outside the $apply(T)$ predicate. To connect the $\vdash$-predicate (primarily designed for having a simple and executable specification of the rule base) with the $\models$-predicate the list of subgoals generated by $\vdash$ has to be extended into a list of subproblems and in this process the hypotheses have to be extended. This task is performed by the *makesubgoals* predicate.

```
⊨(R,π(H==>G,_,_,_),π(H==>G,R,E,T)):-
    functor(R,F,_), rulename(F),
    ( (G=(_ in _);G=(_ < _);G=(_<*_)),⊢(R,H==>G,S),E=axiom;
      ⊢(R,H==>G ext E,S)
    ),!,subgoals(H,S,T).
```

- A user defined tactic is an arbitrary piece of Prolog code, designed to prove a given theorem in an unknown environment, or to generate parameters which may be passed to other tactics. The effect of the application of a user defined tactic on a certain subproblem $H ==> G$ is defined to be the effect of applying this tactic on the top level of a new theorem $H ==> G$, folding the resulting proof tree into a one level structure, and copying this into the original proof tree. [2]

  If the application of a user defined tactic of the form $tactic(U, X)$ is successful, X is bound to the resulting refinement which would have the same effect as the application of the tactic. You can call a tactic simply by giving the corresponding (possibly parametrised) Prolog call U. If you call a user defined tactic in the short version, the resulting refinement is neither stored nor in any way documented. Reproving the proof tree would require in this case fully executing the tactic, i.e. running through the whole search space again. This may cause serious troubles if the tactic has changed in the meantime. Thus you should use the short form only for calling fixed tactics or tactics which have no effect at all. Tactics which do not have any effect at all can be used in connection with the *tacticals* and to build conditional tactics:

```
repeat (\+ goal(_ => void) then intro)
```

[2] This implies that a tactic cannot move above the starting point it was called from

Is an example for a tactic which works like *repeat intro* except that it does not apply *intro* on negations.

If you call a tactic in the long form $tactic(U, X)$ with an unbound variable as second parameter, the tactic is executed and the resulting refinement is stored in the proof tree. But if you use a tactic of the form $tactic(U, X)$ in the *then* or *repeat* branch of a tactical construct, the result of the tactic application can't be stored, because it would be different in different branches. If you use the long form tactic specification on the ground level, it is expanded and stored in the expanded form. In this case reproving would not require a reexecution of the tactic, it would simply apply the resulting refinement directly - so you can save time if the search space is very large. The long form of a tactic application is furthermore an appropriate tool for debugging tactics.

```
⊨(tactic(U,R),π(G,_,_,_),π(G,tactic(U,R),E,S)):-
    var(R),tactic(U,π(G,_,_,_),π(G,R,E,S)),!.
⊨(tactic(U,R),π(G,_,_,_),π(G,tactic(U,R),E,S)):-
    \+ var(R),⊨(R,π(G,_,_,_),π(G,_,E,S)),!.

⊨(U,π(G,_,_,_),π(G,U,E,S)):-
    functor(U,F,_), \+ rulename(F),
    \+ member(F, [then,or,repeat,complete,try,idtac,undo,tactic]),
    tactic(U,π(G,_,_,_),π(G,_,E,S)).



tactic(U,Q,QQ):-
  ϑthm=:N,N=..N1,genint(I),append(N1,[I],N2),NN=..N2, \+ ϑtheorem(NN)=:_,
  ϑpos(NN):=[], ϑtheorem(NN):=Q,
  ϑuniverse=:CU, ϑautotactic=:CA, ϑstatus=:CS,
  defaultautotactic =: DA,
  select(NN),
  ϑuniverse:=CU, ϑautotactic:=CA, ϑstatus:=CS,
  defaultautotactic:=CA,
  (call(U),!,T=succeeds; T=fails),
  defaultautotactic:=DA,
  ϑtheorem(NN)=:Q0, ϑtheorem(NN):=_,
  ϑuniverse=:CU2, ϑautotactic=:CA2, ϑstatus=:CS2,
  select(N),!,
  ϑuniverse:=CU2, ϑautotactic:=CA2, ϑstatus:=CS2,
  T=succeeds,fold(Q0,QQ).



fold(π(P,R,_,_),π(P,idtac,X, [π(P,_,_,_) ext X])):-var(R),!.
```

```
fold(π(P,R,E,[]),π(P,R,E,[])):-!.
fold(π(H==>G,R,E,S),π(H==>G,R0,EE,SS)):-
  fold(E,S,EE,SS,RR),(idtaclist(RR),R0=R;R0=(R then RR)),!.

fold(E,[],E,[],[]).
fold(E,[π(HH==>G,R,E1,S1) ext E0|T],EE,S,[R0|RR]):-
  fold(π(HH==>G,R,E1,S1),π(_,R0,E0,S2)),
  fold(E,T,EE,SS,RR),append(S2,SS,S).
fold(E,[π(HH==>G,R,E1,S1)|T],EE,S,[R0|RR]):-
  fold(π(HH==>G,R,E1,S1),π(_,R0,_,S2)),
  fold(E,T,EE,SS,RR),append(S2,SS,S).



idtaclist([]).
idtaclist([idtac|X]):-idtaclist(X).
```

- Utility predicates needed:

```
⊨′(_,E,S,E,S):-var(S).
⊨′(_,E,[],E,[]).
⊨′([T|TT],E,[π(HH==>G,_,_,_) ext E0|R],EE,Q):- !,
     ⊨(T,π(HH==>G,_,_,_),π(_,_,E0,S1)),
     ⊨′(TT,E,R,EE,S2), append(S1,S2,Q).
⊨′([T|TT],E,[π(HH==>G,_,_,_)|R],EE,Q):- !,
     ⊨(T,π(HH==>G,_,_,_),π(_,_,_,S1)),
     ⊨′(TT,E,R,EE,S2),append(S1,S2,Q).
⊨′(T,E,[π(HH==>G,_,_,_) ext E0|R],EE,Q):-
     copy(T,TT),⊨(T,π(HH==>G,_,_,_),π(_,_,E0,S1)),
     ⊨′(TT,E,R,EE,S2), append(S1,S2,Q).
⊨′(T,E,[π(HH==>G,_,_,_)|R],EE,Q):-
     copy(T,TT),⊨(T,π(HH==>G,_,_,_),π(_,_,_,S1)),
     ⊨′(TT,E,R,EE,S2),append(S1,S2,Q).



subgoals(_,[],[]).
subgoals(H0,[==>G ext T|L],[π(H0==>G,_,_,_) ext T|LL]):-
     !,subgoals(H0,L,LL).
subgoals(H0,[==>G|L],[π(H0==>G,_,_,_)|LL]):-
     !,subgoals(H0,L,LL).
subgoals(H0,[-(TL)==>G ext T|L],[π(HH==>G,_,_,_) ext T|LL]):-
     !,thin_hyps(TL,H0,HH),subgoals(HH,L,LL).
subgoals(H0,[+(TL)==>G ext T|L],[π(HH==>G,_,_,_) ext T|LL]):-
     !,replace_hyps(TL,H0,HH),subgoals(HH,L,LL).
subgoals(H0,[H==>G ext T|L],[π(HH==>G,_,_,_) ext T|LL]):-
     !,append(H0,H,HH),subgoals(H0,L,LL).
subgoals(H0,[H==>G|L],[π(HH==>G,_,_,_)|LL]):-
     !,append(H0,H,HH),subgoals(H0,L,LL).
```

Note the special forms of hypothesis list modifications returnable by a rule: $-(Tlist)$ is a list of hypotheses to be thinned out for the subproblems, $+(Hyps)$ is a list of hypotheses to *replace* those of the same name in the subproblems.

1.9.3 Shorthand Notations

There are several shorthand notations for rules and tacticals, which simply allow you to avoid unnecessary writing of *apply*. There are no shorthand notations for pseudotactics.

```
intro:-apply(intro).
intro(X):-apply(intro(X)).
intro(X,Y):-apply(intro(X,Y)).
intro(X,Y,Z):-apply(intro(X,Y,Z)).


elim(X):-apply(elim(X)).
elim(X,Y):-apply(elim(X,Y)).
elim(X,Y,Z):-apply(elim(X,Y,Z)).


equality:-apply(equality).
equality(X):-apply(equality(X)).


decide(X):-apply(decide(X)).
decide(X,Y):-apply(decide(X,Y)).


simplify:-apply(simplify).


compute(X):-apply(compute(X)).
compute(hyp(X),Y):-apply(compute(hyp(X),Y)).
compute(hyp(X),Y,Z):-apply(compute(hyp(X),Y,Z)).


reduce(X):-apply(reduce(X)).
reduce(X,Y):-apply(reduce(X,Y)).


subst(over(V,X),Y):-apply(subst(over(V,X),Y)).
subst(over(V,X),Y,Z):-apply(subst(over(V,X),Y,Z)).
```

```
seq(X):-apply(seq(X)).
seq(X,Y):-apply(seq(X,Y)).


thin(L) :- apply(thin(L)).


hyp(X):-apply(hyp(X)).


arith:-apply(arith).


unroll(X):-apply(unroll(X)).
unroll(X,Y):-apply(unroll(X,Y)).


T1 then T2:-apply(T1 then T2).

T1 or T2:-apply(T1 or T2).

repeat T:-apply(repeat T).

complete T:-apply(complete(T)).

try T:-apply(try T).

pure(X):-apply(pure(X)).
```

Chapter 2

The Rule Base

This chapter defines the rule base of the type theoretic logic under consideration. The rules are presented exactly in their internal representation. This has the important advantage that there is absolutely no difference between the logic documented and the logic implemented. In fact this document is not only a reference manual, but the precise and executable formal specification of a logical system: at least one step to increase the reliability of theorem proving and more general reasoning systems. Hopefully the use and (proof) reading of this documentation will increase the reliability of the implementation much more than only closed shop debugging and will increase your confidence in the implementation as well.

2.1 Overview

This chapter contains the definition of the $\vdash$-predicate. The $\vdash$-predicate is called with three parameters:

1. The *rule specification* (intro, elim(x), ...), used directly for calling the inference rule. If there are several different inference rules applicable in one situation then these inference rules have different specifications. On the other hand, the rule specification is chosen in such a way that semantically similar rules have the same specification. The rule parameters $at(I)$ and $new[x, ...]$ are in all cases optional. Please pay attention, that new variables have to be supplied always as a list. *new* is defined as key word for reasons of simplicity.

   ```
   ?- op(100,fx,['new']).
   ```

   New variables are always generated from the sequence $v0, v1, v2, ...$ avoiding identifier clashes. The default universe level is taken from the proof tree in a similar way as the autotactic. The specification of a rule may be simplified

using Prolog variables, which is especially valuable in the case of the *subst* and *compute* rules. This approach allows the reusability of rules in a different context. However rules are always called on a copy of the rule specification. As a result variable bindings may only be used inside a single rule specification to formalise structural restriction on the parameters, you will never get the values assigned to the variables outside the $\vdash$-predicate.

2. A pattern of the form H ==> G *ext* E, where H denotes the current *hypothesis list*, G is a pattern for the current *goal* and E describes the *extract term* generated by this inference rule. In the case of membership or equality goals, as well as in goals of the form $A<B$ or $A<= B$, the specification of the extract term is omitted, because in these cases the extract term is always equal to *axiom*. The application of an inference rule is possible only if the current goal matches G. The selection of an inference rule is driven by the rule specification and the structure of the current goal. H refers to a Prolog list of hypotheses, i.e. either definitions, references to other theorems or assumptions of the form $x : h_x$, which you should read as *"let x be an arbitrary element of the type* h_x*"*, where h_x either is a type in the traditional sense or any proposition. In the last case you may think of x as the name of the proposition, but in fact x is an arbitrary element, i.e. a proof of that proposition. The current goal G always is a type, which will be refined during the process of proof. If you think of G as a proposition this refinement yields a proof. If you think of G as a traditional type, this refinement results in an element of that type (program). The extract term E is either a ground term (like *axiom*) or contains variables which are connected to the extract terms provided by the proofs of the subgoals. The extract term is always of the type of the current goal. For each form of terms of the type theoretic language there is at least one rule which produces an extract term of this form.

3. A list of *subgoals*, the elements of which are either of the form H_i ==> G_i, if their proof does not contribute to the extract term of the current inference rule, or of the form H_i ==> G_i *ext*E_i consisting of the subgoal and a variable representing the extract term provided by that subgoal. The inference rules describe only the increment to the hypotheses, i.e. additional properties which are available for the proof of the corresponding subgoals. If there are no additional hypotheses H_i is omitted completely.

The rule base is divided into sections each describing a basic type or a (family of) type constructors (*lists, products, unions,...*). Each section starts with a short motivation and an overview of the intended meaning of that type. The sections are divided into subsections *Type Formation*, *Constructors*, and *Selectors* which consist of the inference rules. Strictly speaking the subsections *Type Formation*

should be considered as part of the *Constructors* for universes. They are distributed only for reasons of better readability. The inference rules itself are classified (as *refinement/realisation rules, membership rules*, or *equality rules*) and informally commented. These comments have no further formal role and can be ignored. The idea behind these comments was to clarify the meaning and possible ways of application of the inference rule to the inexperienced user. If there is a number in front of a rule, this number gives a reference to the corresponding rule in [2] We distinguish the following sorts of rules:

- **Constructor Rules**

  1. **refinement/realisation rules:** They describe the ways for straight forward refinement of the proof. The main result in applying such a rule is the *refinement* of the extract term of the top level goal, corresponding to the refinement step connected with the rule. The extract terms generated by proving the subgoals have a direct influence on the extract term of the refinement rule. Realisation rules describe a the expllicite generation of a realisation of the goal, i.e. they produce a complete extract term.
  2. **membership rules:** They are applicable to goals of the form A *in* T only. Their aim is to prove that the fully refined term A is in fact a member of T. The extract term of such a proof is always *axiom* (and therefore generally omitted in the description of the rules). The membership rules for *universes*, i.e. the rules which apply to goals of the form A *in* $u(i)$, are sometimes also called *well-formedness* rules. To avoid the application of membership rules on equalities the membership rules are (if necessary) guarded by the *noequal* predicate:

     ```
     noequal(_=_):-!,fail.
     noequal(_).
     ```

  3. **equality rules:** They are applicable only to goals of the form $A = B$ *in* T. There is one general equality rule at the end of the rule base which catches all equalities of the form $A = A'$ *in* T where A and A' are syntactically equivalent (more formally speaking A and A' are α-convertible). That is why only non trivial equality rules are given explicitly. There are two types of equality rules: equality rules for types of the form $A = B$ *in* $u(i)$ (in the section *Type Formation*) and equality rules for members of a type T (in the section *Constructors*). Both sections logically should contain the corresponding equality rules. If there is no explicit equality rule, the catch all rule applies.

- **Selector Rules**

1. **refinement rules:** The refinement rules for selectors are *elim* rules or *decide* rules. The *elim* rules exploit the structural properties of the type of a variable in the hypothesis list for generating a *selector* construct which is able to handle the general case. The *decide* rules correspond to decidable basic predicates and yield the appropriate decision operator as extract term. These rules correspond only two partial refinements of the extract terms, therefore there are no realisation rules for selectors.
2. **membership rules:** The membership rules for selectors are the most difficult ones because of the rich type structure. Suppose you have a goal of the form $H ==> sel(E, ...)\ in\ T$, with E being a possibly deeply structured term the type of which, say T' is not obvious, and with T being an instantiation of a type scheme, depending on the current value of E. To deal with this situation, you have to use a *membership rule* of the form $intro(using(T'), over(z, T_z))$, where $using(T')$ reflects your assumptions about the intended semantics of E, and a $over(z, T_z)$ describes a type scheme, which (instantiated with E) would yield T. In most cases there is no direct depency between the type T of the induction term and the current value of the base term E. To simplify the automatic proof of these wellformedness goals there is usually a *derived rule*, which assumes the result type to be constant. If you have really nontrivial wellformedness goals to prove, you should switch the system into the *pure* mode, and supply the type pattern explicitly. But this could be done by a user defined tactic. Still there is the problem of supplying the type information for the base term. For the simplest case, that the base term is a single variable, which is declared in the hypothesis list, there is again a *derived rule* which simply accesses the declaration from the hypothesis list for generating the type of the base term.
3. **equality rules:** The equality rules for selector terms do not describe the equality between to selctor terms, but the possible ways of *reducing* the selector terms to simpler ones. These equality rules are usually specified in the form *reduce(...)* with a goal of the structure $H ==> sel(E, ...) = E'\ in\ T$. From theire nature they should be considered as *conditional rewriting rules*. They are not automatically aplicable because one need a guess, which of the possible preconditions are valid, and therefore in which way the rewriting should work. *Unconditional rewrite rules* are not explicitly given in this chapter, they are available as shorthand notations for *term rewriting rules* (see chapter 5).

There are some rules in the system, which are from their nature efficiency rules. These rules may be deactivated by running the *inference engine* in the *pure* mode. They are marked with a special $\circ$. The *derived* predicate deactivates these rules in

the *pure* mode. The *rulename* predicate is used to shorten the search process in the case of errors. Ensure that the list of rule names is really complete all the time.

```
rulename(X):-
    member(X, [intro, elim, reduce, decide, equality, unroll, hyp, seq,
               lemma, def, subst, compute, normalise, simplify, arith, thin, because]).
```

derived:-ϑ_{status}=:nonpure.

- ⊢(intro,H==>G *ext* X,[]):-
  derived,decl(X:GG,H),α(G,GG).

2.2 Atom

The type *atom* is provided to model character strings. The elements of the type *atom* are arbitrary, possibly empty strings $'...'$, which are (for reasons of syntactical uniqueness) represented in the form $atom('...')$. The equality of the canonical elements of the type *atom* is effectively decidable, i.e. it forms a basic computational operation. Therefore there is a decision operator $atom\_eq(a, b, s, t)$, which is defined to be s if $a = b$ *in* *atom* and t otherwise .

2.2.1 Type Formation

1 ⊢(intro(atom),_==>u(_) *ext* atom,[]).

realisation rule: *atom* is a realisation for any universe

2 ⊢(intro,_==>atom in u(_),[]).

membership rule: *atom* is a type, i.e. member of any universe.

2.2.2 Constructors

3 ⊢(intro(atom(A)),_==>atom *ext* atom(A),[]):-atom(A).

realisation rule: atoms $atom(A)$ are possible realisations for the type *atom*.

4 ⊢(intro,_==>atom(A) in atom,[]):-atom(A).

membership rule: Terms of the form $atom(A)$ are elements of the type *atom*.

2.2.3 Selectors

- ⊢(decide(X=Y in atom,new[U]),
  H==>T *ext* atom_eq(X,Y,σ(E1,[axiom],[U]),σ(E2,[lambda(~,axiom)],[U])),
  [[U:X=Y in atom]==>T *ext* E1,
  [U:X=Y in atom=>void]==>T *ext* E2,
  ==>X in atom,
  ==>Y in atom]):-
  \+ var(X), \+ var(Y),syntax(H,X),syntax(H,Y),free([U],H).

  refinement rule: The decision operator may be used as a refinement for any T, provided both the *then* and the *else* part are refinements of T. This rule gives you the possibility of introducing case analysis in the current proof.

5 ⊢(intro(new[Z]),H==>atom_eq(A,B,X,Y) in T,
  [==>A in atom, ==>B in atom,
  [Z:A=B in atom]==>X in T,
  [Z:A=B in atom=>void]==>Y in T]):-
  free([Z],H).

  membership rule: The decision operator is an element of any type T, provided both the *then* and the *else* part of the decision operator are elements of T.

6 ⊢(reduce(true),_==>atom_eq(A,B,X,Y)=X in T,
  [==>X in T, ==>Y in T, ==>A=B in atom]).
  ⊢(reduce(false),_==>atom_eq(A,B,X,Y)=Y in T,
  [==>X in T, ==>Y in T, ==>A=B in atom => void]).

  equality rules: These rules allow the reduction of the current goal, if you can prove the preconditions necessary for this simplifications.

The type *unary* is provided in order to satisfy a requirement for a type inhabited by only a single term that arises when recursive types are constructed. It is also a convenient way of representing a trivially inhabitted type (and hence true judgement). The only term inhabiting *unary* is *unit*.

2.2.4 Type Formation

1 ⊢(intro(unary),_==>u(_) *ext* unary,[]).

  realisation rule: *unary* is a realisation for any universe

2 ⊢(intro,_==>unary in u(_),[]).

  membership rule: *unary* is a type, i.e. member of any universe.

2.2.5 Constructors

3 ⊢(intro,_==>unary *ext* unit,[]).

realisation rule: *axiom* is a realisation of the type *unary*.

4 ⊢(intro,_==>unit in unary,[]).

membership rule: Terms of the form *unit* are elements of the type *unary*.

2.2.6 Selectors

```
5 ⊢(elim(X), H==>T ext E, [==>TT ext E] ) :-
       decl(X:unary,H),
       s( T, [unit], [X], TT ).
```

2.3 Void

The type *void* is empty, i.e. there are no members of *void*. Assuming the underlying proposition as type paradigm *void* may be regarded as any unprovable proposition. A hypothesis of the form $x : void$ stating that *void* is inhabited by x is the typical form of stating a contradiction. From such a contradiction you can derive any goal. The symbolic extract term $any(x)$ is member of any type, if x is a member of *void*.

2.3.1 Type Formation

1 ⊢(intro(void),_==>u(_) *ext* void,[]).

realisation rule: *void* is a realisation for any universe.

2 ⊢(intro,_==>void in u(_),[]).

membership rule: *void* is a type, i.e. a member of any universe.

2.3.2 Selectors

3 ⊢(elim(X),H==>_ *ext* any(X),[]):- decl(X:void,H).

refinement rule: a hypothesis stating that *void* is inhabited by some X is the incarnation of a contradiction. From such a contradiction you can prove anything. This is reflected in the extract term.

4 ⊢(intro,_==>any(X) in _, [==>X in void]).

membership rule: $any(X)$ is a fictive member of each type indicating which contradiction X was exploited by the proof.

2.4 Pnat

The type *pnat* supplies the natural numbers with Peano arithmetic. The canonical elements of *pnat* are 0 and terms of the form $s(...)$, where s is the successor function on natural numbers. Inductive definition terms have the form $p\_ind(x, a, [u, v, t])$. They define the result of a mapping from *pnat* into some other type T. The terms a and t have to be of the type T, u and v are free variables in t, and it is assumed that u ranges over *pnat* and that v ranges over T. If $x = 0$ *in pnat* the inductive definition term is defined to be equal to a. If x has the form $s(x')$, then the inductive definition term is defined to be equal to the value of s, if u is set to x' and v is set to the value of the inductive definition term for x'. Let us consider two examples.

- The predecessor of y could be defined as $p\_ind(y, 0, [u, \tilde{}, u])$, [1] i.e. it is set to u if y has the form $s(u)$ and is defined to be 0 for 0. Please note, that such an inductive definition term defines only a single value. If you want to describe the predecessor *function* you have to use a λ-construction $lambda(y, p\_ind(y, 0, [u, \tilde{}, u]))$ (see below). Note furthermore, that inductive definition terms, like all other terms, are always *totally defined*. There is no way of specifying partial constructs, you always have to define the value for all cases.

- The embedding of a natural number n in their peano interpretation into the integers is defined by $p\_ind(n, 0, [\tilde{}, u, u + 1])$, i.e. the element 0 of *pnat* is mapped into 0 in *int* and a term of the form $s(i)$ is mapped into $u + 1$, if i is mapped into u.

2.4.1 Type Formation

- ⊢(intro(pnat),_==>u(_) *ext* pnat,[]).

  realisation rule: *pnat* is a realisation for any universe.

- ⊢(intro,_==>pnat in u(_),[]).

  membership rule: *pnat* is a type, i.e. a member of any universe.

2.4.2 Constructors

- ⊢(intro(0),_==>pnat *ext* 0,[]).

  realisation rule: 0 is a realisation for the type *pnat*.

[1] the defining occurrences of bound variables which do not appear in their scope may be omitted, i.e. you can write instead of a senseless variable name a tilde. You should use this convention as far as possible, to avoid unnecessary confusion to other persons, and to improve the systems behaviour, because it is obvious that no substitution has to be carried out in this case.

- ⊢(intro,_==>0 in pnat,[]).

  membership rule: 0 is an element of *pnat*.

- ⊢(intro(s),_==>pnat *ext* s(X), [==>pnat *ext* X]).

  refinement rule: $s(...)$ is a partial refinement for the type *pnat*, leaving the refinement of the predecessor open.

- ⊢(intro,_==>s(X) in pnat, [==>X in pnat]).

  membership rule: $s(X)$ is an element of *pnat*, if X is an element of *pnat*.

- ⊢(intro(s),_==>X=Y in pnat, [==>s(X)=s(Y) in pnat]).

  equality rule: To prove that X and Y are equal in *pnat*, it is sufficient to prove that the successors of X and Y are equal in *pnat*.

2.4.3 Selectors

- ⊢(elim(X,new[U,V]),H==>T *ext* p_ind(X,T0,[U,V,Ts]),
  [==>TT0 *ext* T0,
  [U:pnat,V:TTu]==>Tsu *ext* Ts]):-
  decl(X:pnat,H),free([U,V],H,HH),s(T,[s(U)],[X],Tsu),
  shyp(HH,HH==>T,[0],[X],_==>TT0), shyp(HH,HH==>T,[U],[X],_==>TTu).

  refinement rule: The elimination of an arbitrary variable of the type *pnat* in the hypothesis list, generates an *induction* term, the further refinement of which is determined by the subgoals of the *elim* step.

  ⊢(elim(X,cv,new[U,V,W]),H==> T *ext* cv_ind(X,[W,U,Ts]),
  [[W:pnat,U:(V:{V:pnat \ V <* W}=>Tsv)] ==> Tsw *ext* Ts]) :-
  decl(X:pnat,H),free([U,V,W],H),
  s(T, [V], [X], Tsv),
  s(T, [W], [X], Tsw).

  refinement rule: This rule implements course-of-values induction. Instead of a simple induction hypothesis, an induction hypothesis scheme for the goal with the elim-ed variable replaced by any smaller peano natural number.

- ⊢(intro(over(Z,T),new[U,V]),H==>p_ind(E,T0,[X,Y,Ts]) in Te,
  [==>E in pnat, ==>T0 in To, [U:pnat,V:Tu]==>TTs in Tsu]):-
  τ_{var}(Z),syntax([Z:_|H],T),free([U,V],H),
  s(T,[E],[Z],Te),s(T,[s(U)],[Z],Tsu),
  s(T,[0],[Z],To),s(T,[U],[Z],Tu),s(Ts,[U,V],[X,Y],TTs).

  membership rule: An *induction term* is of a given type T_e if you can supply a type scheme $over(z, T_z)$, such that T_e is an instantiation of that type scheme

for E and the subterms of the induction term can be proven to be in the corresponding instantiations T_o and T_s. To simplify the automatic proof of these wellformedness goals there is a derived rule, which assumes the result type to be constant.

- ⊢(intro(new[U,V]),H==>p_ind(E,T0,[X,Y,Ts]) in T,
  [==>E in pnat, ==>T0 in T, [U:pnat,V:T]==>TTs in T]):-
  derived,free([U,V],H),
  s(Ts,[U,V],[X,Y],TTs).

- ⊢(intro(over(Z,T),new[U,V,W]),H==>cv_ind(E,[X,Y,Ts]) in Te,
  [==>E in pnat,
  [V:(W:{U:pnat \ U <* E}=> Tsw)]==>Tsev in Te]):-
  τ_{var}(Z),syntax([Z:_|H],T),free([U,V,W],H),
  s(T,[E],[Z],Te),
  s(Ts,[E,V], [X,Y], Tsev),
  s(T, [W], [Z], Tsw).

  membership rule: An *induction term* is of a given type T_e if you can supply a type scheme $over(z, T_z)$, such that T_e is an instantiation of that type scheme for E and the subterm of the induction term can be proven to be in the corresponding T_s. To simplify the automatic proof of these wellformedness goals there is a derived rule, which assumes the result type to be constant.

  - ⊢(intro(new[U,V,W]),H==>cv_ind(E,[X,Y,Ts]) in T,
    [==>E in pnat,
    [V:(W:{U:pnat \ U <* E}=> T)]==>Tsev in T]):-
    derived,free([U,V,W],H),
    s(Ts,[E,V], [X,Y], Tsev).

- ⊢(decide(X=Y in pnat,new[U]),
  H==>T *ext* pnat_eq(X,Y,σ(E1,[axiom],[U]),σ(E2,[lambda(~,axiom)],[U])),
  [[U:X=Y in pnat]==>T *ext* E1,
  [U:X=Y in pnat=>void]==>T *ext* E2 ,
  ==>X in pnat,
  ==>Y in pnat]):-
  \+ var(X), \+ var(Y), syntax(H,X),syntax(H,Y),free([U],H).

  refinement rule: The decision operator for natural number equality may be used as a refinement for any type T, provided both, the *then* and the *else* part are refinements of T. This rule gives you the possibility of introducing a case analysis in the current proof.

10 ⊢(intro(new [V]),H==>pnat_eq(A,B,S,T) in TT,
  [==>A in pnat, ==>B in pnat,

```
    [V:A=B in pnat]==>S in TT,
    [V:A=B in pnat=>void]==>T in TT ]):-
  free([V],H).
```

membership rule: The decision operator yields a value of the type T, provided both the *then* and the *else* part of the decision operator are elements of T.

11
```
⊢(reduce(true),_==>pnat_eq(A,B,X,Y)=X in T,
    [ ==>X in T, ==>Y in T, ==>A=B in pnat ]).
⊢(reduce(false),_==>pnat_eq(A,B,X,Y)=Y in T,
    [ ==>X in T, ==>Y in T, ==>A=B in pnat => void ]).
```

equality rules: These rules describe value of the decision term under certain assumptions.

12
```
⊢(reduce,H==>cv_ind(E,[X,Y,T] = Tsev in _),
    [ ==> E in pnat ] ) :-
    free( [R], H ),
    s( T, [E,lambda(R,cv_ind(R,[X,Y,T]))], [X,Y], Tsev ).
```

equality rules: This rule describes the value of the course of values induction term under certain assumptions.

2.5 Pless

pless is a type scheme which is introduced for keeping the theoretical system closed. For each A and B of type *pnat* there exists a type $A< *B$ which is inhabited by *axiom*,if A and B are naturals and A is less then B, or is empty otherwise. In this sense we may assume *axiom* as the only canonical element of $A< *B$.

2.5.1 Type Formation

1 ⊢(intro(<*),_==>u(_) *ext* A<*B, [==>pnat *ext* A, ==>pnat *ext* B]).

refinement rule: $A < *B$ is a refinement for any universe, if A and B are refinements of the type *pnat*.

2 ⊢(intro,_==>A<*B in u(_), [==>A in pnat, ==>B in pnat]).

membership rule: $A < *B$ is a type of any universe level if A and B are elements of the type *pnat*.

2.5.2 Constructors

- ⊢(intro,_==>I<*J,[]):-pnat(I),pnat(J),pless(I,J).

  realisation rule: If I and J are canonical members of *pnat*, i.e. natural numbers, then $I < *J$ is inhabited by *axiom*, if I is less J.

3 ⊢(intro,_==>axiom in (A<*B), [==>A<*B]).

membership rule: Axiom is a canonical member of $A < *B$, if the type $A < *B$ is inhabited at all.

2.5.3 Selectors

4 ⊢(decide(X<*Y,new[U]),
H==>T *ext* pless(X,Y,σ(E1,[axiom],[U]),σ(E2,[lambda(~,axiom)],[U])),
[[U:X<*Y]==>T *ext* E1,
[U:X<*Y=>void]==>T *ext* E2 ,
==> X in pnat,
==> Y in pnat]):-
\+ var(X), \+ var(Y),syntax(H,X),syntax(H,Y),free([U],H).

refinement rule: The decision operator for the strict ordering of naturals may be used as a refinement for any type T, provided both, the *then* and the *else* part are refinements of T. This rule gives you the possibility of introducing a case analysis in the current proof.

11 ⊢(intro(new [V]),H==>pless(A,B,S,T) in TT,
[==>A in pnat, ==>B in pnat,
[V:A<*B]==>S in TT,
[V:(A<*B=>void)]==>T in TT]):-
free([V],H).

membership rule: The decision operator yields a value of the type T, provided both the *then* and the *else* part of the decision operator are elements of T.

14 ⊢(reduce(true),_==>pless(A,B,X,Y)=X in T,
[==>X in T, ==>Y in T, ==>A<*B]).
⊢(reduce(false),_==>pless(A,B,X,Y)=Y in T,
[==>X in T, ==>Y in T, ==>A<*B=>void]).

equality rules: These rules describe value of the decision term under certain assumptions.

2.5.4 Arithmetic

The aim of this section is to provide the rule base for common arithmetical reasoning. Normally these rules will not be used directly, because the built-in tactic *arith* solves many of the standard arithmetic problems. Nevertheless these explicit *arith*-rules are necessary to enable the handling of more difficult arithmetic problems, and these rules provide a proper semantic base for the *arith* tactic.

- ⊢(arith(Y,left),H==>X<*Z, [==>Y<*X=>void, ==>Y<*Z]):-
  syntax(H,Y).
  ⊢(arith(Y,right),H==>X<*Z, [==>X<*Y, ==>Z<*Y=>void]):-
  syntax(H,Y).
  ⊢(arith(<*), _==>X<*Y, [==>A in pnat, ==> B in pnat]) :-
  pdecide(X,Y,less),
  s_subterm(X,A), s_subterm(Y,B).
  ⊢(arith(<*), H==>X<*Y=>void *ext* lambda(V,V),
  [==>A in pnat, ==> B in pnat]) :-
  pdecide(X,Y,R), \+ R = less,
  free([V],H),
  s_subterm(X,A), s_subterm(Y,B).
  ⊢(arith(<*), _==>X<*Y, [==> A<*B]) :-
  s_strip(X<*Y, A<*B).

2.6 Int

The type *int* supplies the common integer arithmetic with ordering $<$. Terms of the form $A<B$ define a new class of propositions. They form a special class of types (see next section). The canonical elements of the type *int* are the integers in the common sense. There are noncanonical constructors for integers: the usual arithmetic operations. The equality of the canonical elements of the type *int* is effective decidable, i.e. it forms a basic computational operation. Therefore there is a decision operator $int\_eq(a, b, s, t)$, which is defined to be s if $a = b$ *in int* and t otherwise . The inductive definition terms have the form $ind(x, [a^-, b^-, t^-], t^0, [a^+, b^+, t^+])$. They define the result of a mapping from *int* into some other type T. The terms t^-, t^0, and t^+ have to be of that type T, a^- and b^- are free variables in t^-, a^+ and b^+ are free variables in t^+, and $a^{\mp}$ is assumed to vary over *int*, whereas $b^{\mp}$ vary over the type T. If x is less than 0, a^- is x, and b^- is the value of that inductive term for $x + 1$, then the value of this induction definition term for x is defined to be t^-. If $x = 0$ *in int* then the value of the inductive definition term is defined to be t^0, and if x if greater than 0, it works the other way round: i.e. if a^+ is x, and b^+ is the value of the inductive definition term for $x - 1$, then the value of the inductive definition term for x is defined to be t^+. Let us consider some examples:

- The *signum* of an integer p could be defined by $ind(p, [\tilde{}, \tilde{}, -1], 0, [\tilde{}, \tilde{}, 1])$;
- 2^i could be described as $ind(i, [\tilde{}, \tilde{}, 0], 1, [\tilde{}, p, 2 * p])$;
- and the *factorial* $n!$ could be defined by $ind(n, [\tilde{}, \tilde{}, 0], 1, [u, f, u * f])$.

Again, these inductive definition terms define only single values, if you want to describe a function, you have to use λ-terms.

2.6.1 Type Formation

1 ⊢(intro(int),_==>u(_) *ext* int,[]).

realisation rule: *int* is a realisation for any universe.

2 ⊢(intro,_==>int in u(_),[]).

membership rule: *int* is a type, i.e. a member of any universe.

2.6.2 Constructors

Constructors for integers are the integer numbers and the usual arithmetic operators. The rules are so obvious, they don't need any further explanation.

3 ⊢(intro(C),_==>int *ext* C,[]):-var(C),C='<integer>';integer(C).

realisation rule:

4 ⊢(intro,_==>C in int,[]):-integer(C).

membership rule:

- ⊢(intro(- ~),_==>int *ext* -T, [==>int *ext* T]).

refinement rule:

5 ⊢(intro,_==> -T in int, [==>T in int]).

membership rule:

- ⊢(intro(- ~),_==> T=TT in int, [==> -T = -TT in int]).

equality rule:

6 ⊢(intro(~ + ~),_==>int *ext* M+N, [==>int *ext* M, ==>int *ext* N]).
⊢(intro(~ - ~),_==>int *ext* M-N, [==>int *ext* M, ==>int *ext* N]).
⊢(intro(~ * ~),_==>int *ext* M*N, [==>int *ext* M, ==>int *ext* N]).
⊢(intro(~ / ~),_==>int *ext* M/N, [==>int *ext* M, ==>int *ext* N]).
⊢(intro(~ mod ~),_==>int *ext* M mod N, [==>int *ext* M, ==>int *ext* N]).

refinement rules:

7 ⊢(intro,_==>M+N in int, [==>M in int, ==>N in int]).
⊢(intro,_==>M-N in int, [==>M in int, ==>N in int]).
⊢(intro,_==>M*N in int, [==>M in int, ==>N in int]).
⊢(intro,_==>M/N in int, [==>M in int, ==>N in int]).
⊢(intro,_==>M mod N in int, [==>M in int, ==>N in int]).

membership rules:

- ⊢(intro(~ + ~),_==>M+N=MM+NN in int,[==>M=MM in int, ==>N=NN in int]).
⊢(intro(~ - ~),_==>M-N=MM-NN in int,[==>M=MM in int, ==>N=NN in int]).
⊢(intro(~ * ~),_==>M*N=MM*NN in int,[==>M=MM in int, ==>N=NN in int]).
⊢(intro(~ / ~),_==>M/N=MM/NN in int,[==>M=MM in int, ==>N=NN in int]).
⊢(intro(~mod~),_==>M mod N=MM mod NN in int,[==>M=MM in int, ==>N=NN in int]).

equality rules:

2.6.3 Selectors

8 ⊢(elim(X,new[U,V,W]),H==>T *ext* ind(X,[U,W,T1],T0,[U,W,T2]),
[[U:int,V:U<0,W:Tsucc]==>Tu *ext* T1,
==>TT *ext* T0,
[U:int,V:0<U,W:Tpred]==>Tu *ext* T2]):-
decl(X:int,H),free([U,V,W],H,HH),s(T,[U],[X],Tu),
shyp(HH,H==>T,[U+1],[X],_==>Tsucc),
shyp(HH,H==>T,[0],[X],_==>TT),

shyp(HH,H==>T,[U-1],[X],_==>Tpred).

refinement rule: The elimination of an arbitrary variable of the type *int* in the hypothesis list, generates an *induction* term, the further refinement of which is determined by the subgoals of the *elim* step.

```
9 ⊢(intro(over(Z,T),new[U,V,W]),
      H==>ind(E,[X1,Y1,T1],T0,[X2,Y2,T2]) in Te,
      [ ==>E in int,
        [U:int,W:U<0,V:Tsucc]==>TT1 in Tu,
        ==>T0 in To,
        [U:int,W:0<U,V:Tpred]==>TT2 in Tu ]):-
      τvar(Z),syntax([Z:_|H],T),free([U,V,W],H),
      s(T,[E],[Z],Te),s(T,[U],[Z],Tu),s(T,[U+1],[Z],Tsucc),
      s(T,[0],[Z],To),s(T,[U-1],[Z],Tpred),
      s(T1,[U,V],[X1,Y1],TT1),s(T2,[U,V],[X2,Y2],TT2).
```

membership rule: An *integer induction term* is of a given type T_e if you can supply a type scheme $over(z, T_z)$, such that T_e is an instantiation of that type scheme for E and the subterms of the induction term can be proven to be in the corresponding instantiations T_o and T_s. In most cases there is no direct depency between the type of the induction term and the current value of the base term of that induction term. To simplify the automatic proof of these wellformedness goals there is a derived rule, which assumes the result type to be constant.

```
  ◦  ⊢(intro(new[U,V,W]),H==>ind(E,[X1,Y1,T1],T0,[X2,Y2,T2]) in T,
       [ ==>E in int,
         [U:int,W:U<0,V:T]==>TT1 in T,
         ==>T0 in T,
         [U:int,W:0<U,V:T]==>TT2 in T ]):-
       derived,free([U,V,W],H),
       s(T1,[U,V],[X1,Y1],TT1),s(T2,[U,V],[X2,Y2],TT2).

12 ⊢(reduce(down),_==>ind(E,[X1,Y1,T1],T0,TT)=TT1 in T,
       [ ==>ind(E,[X1,Y1,T1],T0,TT) in T, ==>E<0 ]):-
       s(T1,[E,ind(E+1,[X1,Y1,T1],T0,TT)],[X1,Y1],TT1).
   ⊢(reduce(base),_==>ind(E,T1,T0,T2)=T0 in T,
       [ ==>ind(E,T1,T0,T2) in T, ==>E=0 in int ]).
   ⊢(reduce(up),_==>ind(E,TT,T0,[X2,Y2,T2])=TT2 in T,
       [ ==>ind(E,TT,T0,[X2,Y2,T2]) in T, ==>0<E ]):-
       s(T2,[E,ind(E-1,TT,T0,[X2,Y2,T2])],[X2,Y2],TT2).
```

equality rules: These rules describe the reduction of the induction terms depending on an assumption about the value of the base term, i.e. whether

thisa term is less than, greater than, or equal to zero.

- ⊢(decide(X=Y in int,new[U]),
  H==>T *ext* int_eq(X,Y,σ(E1,[axiom],[U]),σ(E2,[lambda(~,axiom)],[U])),
  [[U:X=Y in int]==>T *ext* E1,
  [U:X=Y in int=>void]==>T *ext* E2,
  ==>X in int,
  ==> Y in int]):-
  \+ var(X), \+ var(Y), syntax(H,X),syntax(H,Y),free([U],H).

  refinement rule: The decision operator for integer equality may be used as a refinement for any type T, provided both, the *then* and the *else* part are refinements of T. This rule gives you the possibility of introducing a case analysis in the current proof.

10 ⊢(intro(new [V]),H==>int_eq(A,B,S,T) in TT,
  [==>A in int, ==>B in int,
  [V:A=B in int]==>S in TT,
  [V:A=B in int=>void]==>T in TT]):-
  free([V],H).

  membership rule: The decision operator yields a value of the type T, provided both the *then* and the *else* part of the decision operator are elements of T.

13 ⊢(reduce(true),_==>int_eq(A,B,X,Y)=X in T,
  [==>X in T, ==>Y in T, ==>A=B in int]).
  ⊢(reduce(false),_==>int_eq(A,B,X,Y)=Y in T,
  [==>X in T, ==>Y in T, ==>A=B in int => void]).

  equality rules: These rules describe value of the decision term under certain assumptions.

2.7 Less

less is a type scheme which is introduced for keeping the theoretical system closed. For each A and B of type *int* there exists a type $A<B$ which is inhabited by *axiom*,if A and B are integers and A is less then B, or is empty otherwise. In this sense we may assume *axiom* as the only canonical element of $A<B$.

2.7.1 Type Formation

1 ⊢(intro(<),_==>u(_) *ext* A<B, [==>int *ext* A, ==>int *ext* B]).

refinement rule: $A < B$ is a refinement for any universe, if A and B are refinements of the type *int*.

2 ⊢(intro,_==>A<B in u(_), [==>A in int, ==>B in int]).

membership rule: $A < B$ is a type of any universe level if A and B are elements of the type int.

2.7.2 Constructors

- ⊢(intro,_==>I<J,[]):-integer(I),integer(J),I<J.

realisation rule: If I and J are canonical members of int, i.e. integer numbers, then $I< J$ is inhabited by $axiom$, if I is less J.

3 ⊢(intro,_==>axiom in (A<B), [==>A<B]).

membership rule: Axiom is a canonical member of $A<B$, if the type $A<B$ is inhabited at all.

2.7.3 Selectors

4 ⊢(decide(X<Y,new[U]),
H==>T *ext* less(X,Y,σ(E1,[axiom],[U]),σ(E2,[lambda(~,axiom)],[U])),
[[U:X<Y]==>T *ext* E1,
[U:X<Y=>void]==>T *ext* E2,
==>X in int,
==>Y in int]):-
\+ var(X), \+ var(Y),syntax(H,X),syntax(H,Y),free([U],H).

refinement rule: The decision operator for the strict ordering of integers may be used as a refinement for any type T, provided both, the *then* and the *else* part are refinements of T. This rule gives you the possibility of introducing a case analysis in the current proof.

11 ⊢(intro(new [V]),H==>less(A,B,S,T) in TT,
[==>A in int, ==>B in int,
[V:A<B]==>S in TT,
[V:(A<B=>void)]==>T in TT]):-
free([V],H).

membership rule: The decision operator yields a value of the type T, provided both the *then* and the *else* part of the decision operator are elements of T.

14 ⊢(reduce(true),_==>less(A,B,X,Y)=X in T,
[==>X in T, ==>Y in T, ==>A<B]).
⊢(reduce(false),_==>less(A,B,X,Y)=Y in T,
[==>X in T, ==>Y in T, ==>A<B=>void]).

equality rules: These rules describe value of the decision term under certain assumptions.

2.7.4 Arithmetic

The aim of this section is to provide the rule base for common arithmetical reasoning. Normally these rules will not be used directly, because the built-in tactic *arith* solves many of the standard arithmetic problems. Nevertheless these explicit *arith*-rules are necessary to enable the handling of more difficult arithmetic problems, and these rules provide a proper semantic base for the *arith* tactic.

- ⊢(arith(Y,left),H==>X<Z, [==>Y<X=>void, ==>Y<Z]):-
  syntax(H,Y).
  ⊢(arith(Y,right),H==>X<Z, [==>X<Y, ==>Z<Y=>void]):-
  syntax(H,Y).

- ⊢(arith(+,Z),_==>X<Y, [==>X+Z<Y+Z]).
  ⊢(arith(-,Z),_==>X<Y, [==>X-Z<Y-Z]).
  ⊢(arith(*,0<Z),_==>X<Y, [==>X*Z<Y*Z, ==>0<Z]).
  ⊢(arith(*,Z<0),_==>X<Y, [==>Y*Z<X*Z, ==>Z<0]).

- ⊢(arith(+),_==>0<X*Y, [==>0<X, ==>0<Y]).
  ⊢(arith(-),_==>0<X*Y, [==>X<0, ==>Y<0]).

- ⊢(arith,_==>X mod Y<0=>void, [==>X in int, ==>Y in int]).
  ⊢(arith(+),_==>X mod Y<Y, [==>X in int, ==>0<Y]).
  ⊢(arith(-),_==>X mod Y<(-Y), [==>X in int, ==>Y<0]).

2.8 List Types

The type of all finite lists over a given type A is written in the form $A\ list$. The elements of the list type $A\ list$ are the empty list nil and nonempty lists $x::y$ constructed from an element x of the base type A and a list y of the type $A\ list$. Examples for members of the type $int\ list$ would be nil, $1::nil$, $2::3::nil$, etc. There is one universal list destructor, the list induction term $list\_ind(x, y, [u, v, w, z])$, it maps the list x of type $A\ list$ into some other type T, provided that y, and z are from the type T. u, v and w are free variables in z which become bound in the list induction term. If x is the empty list, the list induction term ist defined to be y. If x is a nonempty list and therefore has the form $u :: v$, and if furthermore w is assumed to be the value of the list induction term for v, then the value of the list induction term for x is defined to be z. Let us consider the following examples:

- The length of a list l is described by $list\_ind(l, 0, [\tilde{}, \tilde{}, n, n + 1])$.
- Suppose x is of the type $A\ list$ and there is a special element err [2] in A, then head and the tail of the list x can be described as $list\_ind(x, err, [h, \tilde{}, \tilde{}, h])$, and $list\_ind(x, nil, [\tilde{}, t, \tilde{}, t])$, respectively.
- The *sum* of the elements of a list k of integers could be described in the form $list\_ind(k, 0, [h, \tilde{}, s, h + s])$.

2.8.1 Type Formation

1 ⊢(intro(~ list),_==>u(l) *ext* A list, [==>u(l) *ext* A]).

refinement rule: *A list* is a refinement for any universe, if A is a refinement for the same universe. This is a typical stepwise refinement rule. You make a partial decision (to use a list structure), but let the base type of the list still open. So there is a subgoal still stating that your initial universe is inhabited, which would result (during further refinement) in some type A.

2 ⊢(intro,_==>A list in u(l), [==>A in u(l)]).

membership rule: *A list* is a type of universe level i, i.e. a member of $u(i)$ if A is a type of universe level i.

- ⊢(intro,_==>A list=B list in u(l), [==>A=B in u(l)]).

equality rule: Two list types are equal if the corresponding base types are equal.

[2] Remember, that there is no way of describing partial constructs, you always need an *error element* or some equivalent.

2.8.2 Constructors

3 ⊢(intro(*at*(I),nil),_==>A list *ext* nil,[==>A list in u(I)]):- level(I).

realisation rule: *nil* is a realisation for any well-formed list type.

4 ⊢(intro(*at*(I)),_==>nil in A list,[==>A list in u(I)]):- level(I).

membership rule: *nil* is an element of any well-formed list type.

5 ⊢(intro(::),_==>A list *ext* B::C, [==>A *ext* B, ==>A list *ext* C]).

refinement rule: List construction is a partial refinement over any list type. The head and the tail of the list are derived as refinement of the basic type or of the list type again.

6 ⊢(intro,_==>B::C in A list, [==>B in A, ==>C in A list]).

membership rule: A list construction term $B :: C$ is an element of the type *Alist*, if the head B is an element of the basic type A and the tail C is an element of the list type *Alist* again.

- ⊢(intro,_==>A::B=AA::BB in T list, [==>A=AA in T, ==>B=BB in T list]).

equality rule: Two nonempty lists are equal if the heads and tails are equal in the corresponding types.

2.8.3 Selectors

7 ⊢(elim(X,new[U,V,W]),H==>T *ext* list_ind(X,Tb,[U,V,W,Tu]),
[==>Tnil *ext* Tb,
[U:A,V:A list,W:Tv]==>Tuv *ext* Tu]):-
decl(X:A list,H), free([U,V,W],H,HH), s(T,[V],[X],Tv),
shyp(HH,H==>T,[nil],[X],_==>Tnil),
shyp(HH,H==>T,[U::V],[X],_==>Tuv).

refinement rule: The elimination of an arbitrary variable of a list type *A list* in the hypothesis list, generates an *list induction* term, the further refinement of which is determined by the subgoals of the *elim* step.

8 ⊢(intro(using(A list),over(Z,T),new[U,V,W]),
H==>list_ind(E,Tbase,[X,Y,K,Tind]) in Te,
[==>E in A list,
==>Tbase in Tnil,
[U:A,V:A list,W:Tv]==>Tstep in Tuv]):-
syntax(H,A),τ_{var}(Z),syntax([Z:_|H],T),free([U,V,W],H),
s(T,[E],[Z],Te),s(T,[U::V],[Z],Tuv),
s(T,[nil],[Z],Tnil),s(T,[V],[Z],Tv),

s(Tind,[U,V,W],[X,Y,K],Tstep).

membership rule: A *list induction term* is of a given type T_e if you can supply a type scheme $over(z, T_z)$, such that T_e is an instantiation of that type scheme for E and the subterms of the induction term can be proven to be in the corresponding instantiations T_o and T_s, and if you can predict the type of the base term $using(A\ list)$. In most cases there is no direct depency between the type of the induction term and the current value of the base term of that induction term. To simplify the automatic proof of these wellformedness goals there is a *derived rule*, which assumes the result type to be constant. Still there is the problem of supplying the type information for the base term. For the simplest case, that the base term is a single variable, which is declared in the hypothesis list, there is again a *derived rule* which simply accesses the declaration from the hypothesis list for generating the type of the base term.

- ⊢(intro(using(A list),new[U,V,W]),
  H==>list_ind(E,Tbase,[X,Y,Z,Tind]) in T,
  [==>E in A list,
  ==>Tbase in T,
  [U:A,V:A list,W:T]==>Tstep in T]):-
  derived,syntax(H,A),free([U,V,W],H),
  s(Tind,[U,V,W],[X,Y,Z],Tstep).
- ⊢(intro(new[U,V,W]),
  H==>list_ind(E,Tbase,[X,Y,Z,Tind]) in T,
  [==>E in A list,
  ==>Tbase in T,
  [U:A,V:A list,W:T]==>Tstep in T]):-
  derived,type(E,H,A list), free([U,V,W],H),
  s(Tind,[U,V,W],[X,Y,Z],Tstep).

2.9 Disjoint Union Types

If A and B are types, then the disjoint union of A and B, denoted by $A\backslash B$, is also a type. The elements of the disjoint union $A\backslash B$ have the form $inl(x)$, where x is an element of A, or $inr(y)$, where y is an element of B. Whether an element of a disjoint union type is a projection from the left or the right member type of the union is a decidable property, which forms a basic computational operation. The decision operator has the form $decide(z, [u, f], [v, g])$ and yields the value $f_{[x/u]}$ if z has the form $inl(x)$ or $g_{[y/v]}$ if z is of the form $inr(y)$.

2.9.1 Type Formation

1 ⊢(intro(˜ \ ˜),_==>u(I) *ext* A\B, [==>u(I) *ext* A, ==>u(I) *ext* B]).

refinement rule: $A\backslash B$ is a refinement for any universe, if A and B are refinements for the same universe. This is a stepwise refinement rule. You make a partial decision (to use a disjoint union), but let the base types of the union still open. So there are two subgoals still stating that your initial universe is inhabited, which would result (during further refinement) in some types A and B.

2 ⊢(intro,_==>(A\B) in u(I), [==>A in u(I), ==>B in u(I)]).

membership rule: $A\backslash B$ is a type of universe level i, i.e. a member of $u(i)$ if A and B are types of universe level i.

- ⊢(intro,_==>(A\B)=(AA\BB) in u(I),
  [==>A=AA in u(I), ==>B=BB in u(I)]).

  equality rule: Union types are equal if the corresponding base types are equal.

2.9.2 Constructors

3 ⊢(intro(*at*(I),left),_==>A\B *ext* inl(X),
  [==>A *ext* X, ==>B in u(I)]):-
  level(I).

refinement rule: A left injection term $inl(X)$ is a refinement for the union type $A\backslash B$ at any universe level if X is a refinement of the type A and the other half of the union type is well-formed at the universe level given.

4 ⊢(intro(*at*(I)),_==>inl(X) in (A\B),
  [==>X in A, ==>B in u(I)]):-
  level(I).

membership rule: A left injection term $inl(X)$ is a member of the union type $A\backslash B$ at any universe level if X is a member of A and the other half of the union type is well-formed at the universe level given.

- ⊢(intro(*at*(I)),_==>inl(X)=inl(XX) in (A\B),
  [==>X=XX in A, ==>B in u(I)]):-
  level(I).

  equality rule: Left injection terms are equal if the injecting terms are equal, and the other half of the union type is well-formed at the universe level given.

5 ⊢(intro(*at*(I),right),_==>A\B *ext* inr(X),
[==>B *ext* X, ==>A in u(I)]):-
level(I).

refinement rule: A right injection term $inr(X)$ is a refinement for the union type $A\backslash B$ at any universe level if X is a refinement of the type B and the other half of the union type is well-formed at the universe level given.

6 ⊢(intro(*at*(I)),_==>inr(X) in (A\B),
[==>X in B, ==>A in u(I)]):-
level(I).

membership rule: A right injection term $inr(X)$ is a member of the union type $A\backslash B$ at any universe level if X is a member of B and the other half of the union type is well-formed at the universe level given.

• ⊢(intro(*at*(I)),_==>inr(X)=inr(XX) in (A\B),
[==>A in u(I), ==>X=XX in B]):-
level(I).

equality rule: Right injection terms are equal if the injecting terms are equal and the other half of the union type is well-formed at the universe level given.

2.9.3 Selectors

7 ⊢(elim(V,new[X,Y,N1,N2]),H==>T *ext* decide(V,[X,Tx],[Y,Ty]),
[[X:A,N1:V=inl(X) in (A\B)]==>Tinl *ext* Tx,
[Y:B,N2:V=inr(Y) in (A\B)]==>Tinr *ext* Ty]):-
decl(V:A\B,H), free([X,Y,N1,N2],H,HH),
shyp(HH,H==>T,[inl(X)],[V],_==>Tinl),
shyp(HH,H==>T,[inr(Y)],[V],_==>Tinr).

refinement rule: The elimination of an arbitrary variable of a disjoint union type generates to subgoals, one for the case the value of the variables comes from the left base type, and one for the other case. The extract term combines the resulting terms from both cases into a *decision term*.

8 ⊢(intro(using(A\B),over(Z,T),new[U,V,W]),
H==>decide(E,[X,Tx],[Y,Ty]) in Te,
[==>E in (A\B),
[U:A,W:E=inl(U) in A\B]==>Tu in Tleft,
[V:B,W:E=inr(V) in A\B]==>Tv in Tright]):-
syntax(H,A),syntax(H,B),τ_{var}(Z),syntax([Z:_|H],T),free([U,V,W],H),
s(Tx,[U],[X],Tu),s(Ty,[V],[Y],Tv),
s(T,[E],[Z],Te),s(T,[inl(U)],[Z],Tleft),s(T,[inr(V)],[Z],Tright).

membership rule: A *decision term* is of a given type T_e if you can supply a type scheme $over(z, T_z)$, such that T_e is an instantiation of that type scheme for E and the subterms of the decision term can be proven to be in the corresponding instantiations T_{left} and T_{right}, and if you can predict the type of the base term $using(A\backslash B)$. In most cases there is no direct depency between the type of the decision term and the current value of the base term of that decision term. To simplify the automatic proof of these wellformedness goals there is a *derived rule*, which assumes the result type to be constant. Still there is the problem of supplying the type information for the base term. For the simplest case, that the base term is a single variable, which is declared in the hypothesis list, there is again a *derived rule* which simply accesses the declaration from the hypothesis list for generating the type of the base term.

- ⊢(intro(using(A\B),new[U,V,W]),
  H==>decide(E,[X,Tx],[Y,Ty]) in T,
  [==>E in (A\B),
  [U:A,W:E=inl(U) in A\B]==>Tu in T,
  [V:B,W:E=inr(V) in A\B]==>Tv in T]):-
  derived,syntax(H,A),syntax(H,B),free([U,V,W],H),
  s(Tx,[U],[X],Tu),s(Ty,[V],[Y],Tv).
- ⊢(intro(new [U,V,W]),
  H==>decide(E,[X,Tx],[Y,Ty]) in T,
  [==>E in (A\B),
  [U:A,W:E=inl(U) in A\B]==>Tu in T,
  [V:B,W:E=inr(V) in A\B]==>Tv in T]):-
  type(E,H,A\B),
  derived,free([U,V,W],H),
  s(Tx,[U],[X],Tu),s(Ty,[V],[Y],Tv).

2.10 Function Types

Function types $A{\rightarrow}B$ describe mappings from the type A into the type B. The elements of a function type are λ-terms of the form $lambda(x, t_x)$. We may consider function type as logical implications, i.e. a special form of propositions. Propositions are inhabited by their proofs. A proof of a implication $A{\rightarrow}B$ consists in the intuitionistic framework of a mapping, which maps any proof of A into a proof of B, i.e. any member of A into B. There is one selector defined on function types: the *function application f of a* where f is any element of $A{\rightarrow}B$ and a is an element of A.

2.10.1 Type Formation

3 ⊢(intro(˜ => ˜),_==>u(I) *ext* A=>B, [==>u(I) *ext* A, ==>u(I) *ext* B]).

refinement rule: $A \to B$ is a refinement for any universe, if A and B are refinements for the same universe. This is a stepwise refinement rule. You make a partial decision (to use a function type), but let the base types of the mapping still open. So there are two subgoals still stating that your initial universe is inhabited, which would result (during further refinement) in some types A and B.

4 ⊢(intro(new[Y]),H==>(A=>B) in u(I),
[==>A in u(I), [Y:A]==>B in u(I)]):-
free([Y],H).

membership rule: $A \to B$ is a type of universe level i, i.e. a member of $u(i)$ if A is a type of universe level i, and B can be proven to be a type under the assumption of A being inhabited by an arbitrary element Y.

• ⊢(intro,_==>(A=>B)=(AA=>BB) in u(I),
[==>A=AA in u(I), ==>B=BB in u(I)]).

equality rule: Function Types are equal if the corresponding domain and range types are equal.

2.10.2 Constructors

5 ⊢(intro(*at*(I),new[Y]),H==>A=>B *ext* lambda(Y,F),
[[Y:A]==>B *ext* F, ==>A in u(I)]):-
free([Y],H),level(I).

refinement rule: A λ-term $lambda(Y, F)$ is a refinement for the type $A \to B$ at any universe level if F is a refinement of B under the assumption of Y being an arbitrary element of A, and if the domain type A is a member of that given universe level.

6 ⊢(intro(*at*(I),new[Z]),H==>lambda(X,F) in (A=>B),
[[Z:A]==>Fz in B, ==>A in u(I)]):-
free([Z],H),s(F,[Z],[X],Fz),level(I).
⊢(intro(*at*(I)),H==>lambda(X,F) in (A=>B),
[[X:A]==>F in B, ==>A in u(I)]):-
free([X],H),level(I).

membership rule: A λ-term $lambda(X, F)$ is an element of the type $A \to B$ at any universe level if $F_{[z/x]}$ is a member of B under the assumption of z being an arbitrary element of A, and if the domain type A is a member of that given universe level.

```
⊢(intro(at(I),using(AA=>BB),new[X,Y,W]),H==>F in (A=>B),
    [ [X:A,Y:A,W:X=Y in A]==>F of X =F of Y in B,
      ==>A in u(I),
      ==>F in (AA=>BB) ]):-
    free([X,Y,W],H),level(I).
```

- ```
  ⊢(intro(at(I),new[Y]),H==>lambda(Xf,F)=lambda(Xg,G) in (A=>B),
      [ [Y:A]==> FF=GG in B,==>A in u(I) ]):-
      free([Y],H),s(F,[Y],[Xf],FF),s(G,[Y],[Xg],GG),level(I).
  ```

  equality rule: Two λ-terms are equal if you can prove the equality of the base terms under the assumption of the same free variable.

2.10.3 Selectors

8
```
⊢(elim(F,new[Y]),H==>T ext σ(E,[F of X],[Y]),
    [ ==>A ext X, [Y:B]==>T ext E ]):-
    ( decl(F:A=>B,H);
      (decl(F:(V:A=>B),H),\+freevarinterm(B,V))
    ),
    free([Y],H).
⊢(elim(F,on(X),new[Y,Z]),H==>T ext σ(E,[F of X],[Y]),
    [ ==>X in A,
      [Y:B,Z:Y=F of X in B]==>T ext E ]):-
    ( decl(F:(A=>B),H);
      (decl(F:(V:A=>B),H),\+freevarinterm(B,V))
    ),
    free([Y,Z],H),
    syntax(H,X).
```

refinement rule: The elimination of an arbitrary variable of a function type generates a function application term, which is automatically substituted for all occurences of references Y to the instantiation of the range type (proposition) of that function type in the extract term of the proof of the subgoal. The second *elim* rule describes the situation, where you wish to refer to the value of the function application term in further proof.

9
```
⊢(intro(using(A=>B),new[U]),H==>F of Y in T,
    [ ==>F in (A=>B),
      ==>Y in A,
      [U:B]==>U in T]):-
    syntax(H,A),syntax(H,B),free([U],H).
```

membership rule: A *function application term* is of a given type T if you can supply function type for the *base term*, such that F is of that function
```

type, that the argument is in the domain of that function type, and that the memberhood in the range of the function type implies memberhood in T. Still there is the problem of supplying the type information for the function itself. For the simplest case, that the function term is a single variable, which is declared in the hypothesis list, there is again a *derived rule* which simply accesses the declaration from the hypothesis list for generating the type of the base term.

```
○   ⊢(intro(new[U]),H==>F of Y in T,
        [ ==>F in (A=>B),
          ==>Y in A,
          [U:B]==>U in T ]):-
        derived,free([U],H),type(F,H,A=>B).

⊢(equality(new[Y]),H==>F=G in (A=>B),
    [ [Y:A]==> F of Y=G of Y in B,
      ==>F in (A=>B),
      ==>G in (A=>B) ]):-
    free([Y],H).
```

equality rule: The equality rule for function terms differs a little from the other rules, because there are not only syntactic properties which could be used for proving the equality of two functions. This rule describes the *extensionality* of the equality of functions.

2.11 Dependent Function Types

Dependent function types $X : A \rightarrow B$, where A is a type and B denotes a family $B_{X \in A}$ of types, are a generalisation of function types. They describe mappings from the type A into a family of types $B_{a \in A}$, such that an element $X \in A$ is mapped into an element of the type $B_{[X/a]}$. The elements of a dependent function type are also λ-terms of the form $lambda(x, t_x)$, but the result type of t_x may depend on the value of x. One can consider dependent function type from the logical point of view as universally quantified propositions, stating that B is valid for all X of type A. A proof of that universally quantified proposition consists in the intuitionistic framework of a mapping which maps any element X of A into a proof of B, i.e. any member X of A into a member of $B_{[X/a]}$. There is one selector defined on dependent function types: the *function application* $f\ of\ a$ where f is any element of $x : A \rightarrow B$ and a is an element of A.

2.11.1 Type Formation

1 ⊢(intro(X:A=> ~),H==>u(I) *ext* X:A=>B,
[==>A in u(I), [X:A]==>u(I) *ext* B]):-
syntax(H,A),free([X],H).

refinement rule: $X : A \rightarrow B$ is a refinement for any universe, if A is a type, and B is a refinement of the same universe under the assumption of the existence of some X of type A. This is a stepwise refinement rule. You make a partial decision (to use a function type), but let the base types of the mapping still open. So there are two subgoals still stating that your initial universe is inhabited, which would result (during further refinement) in some types A and B.

2 ⊢(intro(new[Y]),H==>(X:A=>B) in u(I),
[==>A in u(I), [Y:A]==>By in u(I)]):-
free([Y],H),s(B,[Y],[X],By).

membership rule: $X : A \rightarrow B$ is a type of universe level i, i.e. a member of $u(i)$ if A is a type of universe level i, and $B[Y/X]$ can be proven to be a type under the assumption of A being inhabited by an arbitrary element Y.

• ⊢(intro(new[Y]),H==>(X:A=>B)=(XX:AA=>BB) in u(I),
[==>A=AA in u(I), [Y:A]==>By=BBy in u(I)]):-
free([Y],H),s(B,[Y],[X],By), s(BB,[Y],[XX],BBy).

equality rule: Dependent function types are equal if the corresponding domain types are equal and if the range types can be proven equal under the assumption of a common free variable Y.

2.11.2 Constructors

5 ⊢(intro(*at*(I),new[Y]),H==>X:A=>B *ext* lambda(Y,F),
[[Y:A]==>By *ext* F, ==>A in u(I)]):-
((free([X],H),By=B,Y=X);
(free([Y],H),s(B,[Y],[X],By))
),
!,level(I).

refinement rule: A λ-term $lambda(Y, F)$ is a refinement for the type $X : A \rightarrow B$ at any universe level if F is a refinement of $B_{[Y/X]}$ under the assumption of Y being an arbitrary element of A, and if the domain type A is a member of that given universe level. The first disjunct describes the special case where X is a free variable in H, and may be used as a preferred variable identifier instead of Y. This has no influence on the logical interpretation, but allows you to exploit the intensional meaning of the variable identifier in further proof.

There is a derived rule which handles the special case of having propositions universally quantified (i.e. parametrised) over some type and automatically supplies a suitable universe level.

- ⊢(intro(new[Y]),H==>X:u(I)=>B *ext* lambda(Y,F),
  [[Y:u(I)]==>By *ext* F, ==>u(I) in u(J)]):-
  derived, free([Y],H),
  s(B,[Y],[X],By),level(I),J is I+1.
  ⊢(intro,H==>X:u(I)=>B *ext* lambda(X,F),
  [[X:u(I)]==>B *ext* F, ==>u(I) in u(J)]):-
  derived, free([X],H),level(I),J is I+1.

6 ⊢(intro(*at*(I),new[Z]),H==>lambda(X,F) in (Y:A=>B),
[[Z:A]==>Fz in Bz, ==>A in u(I)]):-
free([Z],H),s(B,[Z],[Y],Bz),s(F,[Z],[X],Fz), level(I).

membership rule: A λ-term $lambda(X, F)$ is an element of the type $Y{:}A{\rightarrow}B$ at any universe level if $F_{[Z/X]}$ is a member of $B_{[Z/Y]}$ under the assumption of Z being an arbitrary element of A, and if the domain type A is a member of that given universe level.

- ⊢(intro(*at*(I),new [Y]),H==>lambda(Xf,F)=lambda(Xg,G) in (X:A=>B),
  [[Y:A]==> FF=GG in By,==> A in u(I)]):-
  free([Y],H),s(F,[Y],[Xf],FF),s(G,[Y],[Xg],GG),s(B,[Y],[X],By),level(I).

  equality rule: Two λ-terms are equal if you can prove the equality of the base terms under the assumption of the same free variable in the common range type.

- ⊢(intro(*at*(I),using(U:AA=>BB),new[X,Y,W]),
  H==>F in (A=>B),
  [[X:A,Y:A,W:X=Y in A]==>F of X =F of Y in B,
  ==>A in u(I),
  ==>F in (U:AA=>BB)]):-
  free([X,Y,W],H),level(I).

2.11.3 Selectors

7 ⊢(elim(F,on(X),new[Y]),H==>T *ext* σ(E,[F of X],[Y]),
[==>X in A,
[Y:Bx]==>T *ext* E]):-
decl(F:(V:A=>B),H),free([Y],H),syntax(H,X),s(B,[X],[V],Bx).
⊢(elim(F,on(X),new[Y,Z]),H==>T *ext* σ(E,[F of X],[Y]),

```
    [ ==>X in A,
      [Y:Bx,Z:Y=F of X in Bx]==>T ext E ]):-
    decl(F:(V:A=>B),H),free([Y,Z],H),syntax(H,X),s(B,[X],[V],Bx).
```

refinement rule: The elimination of an arbitrary variable of a function type generates a function application term, which is automatically substituted for all occurences of references Y to the instantiation of the range type (proposition) of that function type in the extract term of the proof of the subgoal. The second *elim* rule describes the situation, where you wish to refer to the value of the function application term in further proof.

```
9 ⊢(intro(using(X:A=>B),over(Z,T),new[U,V]),H==>F of Y in Ty,
      [ ==>F in (X:A=>B), ==>Y in A, [U:A,V:Bu]==>V in Tu ]):-
      syntax(H,A),τvar(X),syntax([X:_|H],B),τvar(Z),syntax([Z:_|H],T),
      free([U,V],H),
      s(T,[Y],[Z],Ty),s(B,[U],[X],Bu),s(T,[U],[Z],Tu).
```

membership rule: A *function application term* is of a given type T_e if you can supply a type scheme $over(z, T_z)$, such that T_y is an instantiation of that type scheme for the argument Y and that in general, that for any U in the domain type A and any value V of the range type B_u is in the instantiation of T for U. In most cases there is no direct depency between the type of the function application term and the current value of the argument term. To simplify the automatic proof of these wellformedness goals there is a *derived rule*, which assumes the result type to be constant. Still there is the problem of supplying the type information for the function itself. For the simplest case, that the function term is a single variable, which is declared in the hypothesis list, there is again a *derived rule* which simply accesses the declaration from the hypothesis list for generating the type of the base term.

- ```
  ⊢(intro(using(X:A=>B),new[U,V]),H==>F of Y in T,
      [ ==>F in (X:A=>B), ==>Y in A, [U:A,V:Bu]==>V in T ]):-
      syntax(H,A),syntax([X:_|H],B),τvar(X),
      free([U,V],H),s(B,[U],[X],Bu).
  ```
- ```
  ⊢(intro(new[U,V]),H==>F of Y in T,
      [ ==>F in (X:A=>B), ==>Y in A, [U:A,V:Bu]==>V in T ]):-
      derived,free([U,V],H),type(F,H,X:A=>B),
      s(B,[U],[X],Bu).
  ```

```
10 ⊢(equality(new[Y]),H==>F=G in (X:A=>B),
      [ [Y:A]==> F of Y=G of Y in By,
        ==>F in (X:A=>B),
        ==>G in (X:A=>B) ]):-
```

free([Y],H),s(B,[Y],[X],By).

equality rule: The equality rule for function terms differs a little from the other rules, because there are not only syntactic properties which could be used for proving the equality of two functions. This rule describes the *extensionality* of the equality of functions.

2.12 Product Types

Cartesian product types $A\#B$ consist of ordered pairs $x\&y$ of elements x of A and y of B. We can interpret product types as logical conjunctions, i.e. a special form of propositions. Elements of such a product type would be the proofs of the conjunction. A proof of a conjunction $A\#B$ consists of a proof of A and a proof of B. So we might consider the set of all proofs of $A\#B$ as the set of ordered pairs of proofs of A and B. There is one selector construct *spread*, a generalisation of the usual projection operators: supposed $s = x\&y$ *in* $A\#B$, then $spread(s, [u, v, t])$ is defined to be $t_{[x,y/u,v]}$, i.e. t with u and v substituted by the left and right component of s respectively. The left projection and the right projection operators could be described as $spread(s, [u, \tilde{}, u])$ and $spread(s, [\tilde{}, v, v])$, respectively.

2.12.1 Type Formation

3 ⊢(intro(˜ # ˜),_==>u(l) *ext* A#B, [==>u(l) *ext* A, ==>u(l) *ext* B]).

refinement rule: $A\#B$ is a refinement for any universe, if A and B are refinements for the same universe. This is a stepwise refinement rule. You make a partial decision (to use a Cartesian product), but let the base types of the product still open. So there are two subgoals still stating that your initial universe is inhabited, which would result (during further refinement) in some types A and B.

4 ⊢(intro,_==>(A#B) in u(l), [==>A in u(l), ==>B in u(l)]).

membership rule: $A\#B$ is a type of universe level i, i.e. a member of $u(i)$ if A and B are types of universe level i.

- ⊢(intro,_==>(A#B)=(AA#BB) in u(l),
  [==>A=AA in u(l), ==>B=BB in u(l)]).

  equality rule: Product types are equal if the corresponding base types are equal.

2.12.2 Constructors

7 ⊢(intro,_==>A#B *ext* F&G, [==>A *ext* F, ==>B *ext* G]).

refinement rule: Ordered pairs $F\&G$ are refinements for Cartesian product types $A\#B$ if F is a refinement of A and G is a refinement of B.

8 ⊢(intro,_==>(F&G)in (A#B), [==>F in A, ==>G in B]).

membership rule: An ordered pair $F\&G$ is a members of the Cartesian product type $A\#B$ if F is a member of A and G is a member of B.

- ⊢(intro,_==>U&V=UU&VV in (A#B), [==>U=UU in A, ==>V=VV in B]).

equality rule: Pairs are equal in the product type, if the components are equal in the corresponding base types.

2.12.3 Selectors

9 ⊢(elim(Z,new[U,V,W]),H==>T *ext* spread(Z,[U,V,S]),
[[U:A,V:B,W:Z=U&V in (A#B)]==>Tuv *ext* S]):-
decl(Z:A#B,H),free([U,V,W],H),
s(T,[U&V],[Z],Tuv).

refinement rule: The elimination of an arbitrary variable of a Cartesian product type in the hypothesis list, generates an *spread* term, the further refinement of which is determined by the subgoal of the *elim* step.

10 ⊢(intro(using(A#B),over(Z,T),new[U,V,W]),H==>spread(E,[X,Y,S]) in Te,
[==>E in (A#B), [U:A,V:B,W:E=U&V in (A#B)]==>Suv in Tuv]):-
τ_{var}(Z),syntax([Z:_|H],T),syntax(H,A),syntax(H,B),free([U,V,W],H),
s(S,[U,V],[X,Y],Suv),s(T,[U&V],[Z],Tuv),s(T,[E],[Z],Te).

membership rule: A *spread term* is of a given type T_e if you can supply a type scheme $over(z, T_z)$, such that T_e is an instantiation of that type scheme for the base term E and the subterm of the spread term can be proven to be in the corresponding instantiation $T_{u,v}$, and if you can predict the type of the base term $using(A\#B)$. In most cases there is no direct depency between the type of the decision term and the current value of the base term of that decision term. To simplify the automatic proof of these wellformedness goals there is a *derived rule*, which assumes the result type to be constant. Still there is the problem of supplying the type information for the base term. For the simplest case, that the base term is a single variable, which is declared in the hypothesis list, there is again a *derived rule* which simply accesses the declaration from the hypothesis list for generating the type of the base term.

- ⊢(intro(using(A#B),new[U,V,W]),H==>spread(E,[X,Y,S]) in T,
  [==>E in (A#B), [U:A,V:B,W:E=U&V in (A#B)]==>Suv in T]):-
  derived,syntax(H,A),syntax(H,B),free([U,V,W],H),
  s(S,[U,V],[X,Y],Suv).
- ⊢(intro(new[U,V,W]),H==>spread(E,[X,Y,S]) in T,
  [==>E in (A#B), [U:A,V:B,W:E=U&V in (A#B)]==>Suv in T]):-
  derived,type(E,H,A#B), free([U,V,W],H),
  s(S,[U,V],[X,Y],Suv).

2.13 Dependent Product Types

Dependent product types $x : A\#B$, where A is a type and B denotes a family of types $B_{x \in A}$, are a generalisation of Cartesian product types. They consist of ordered pairs $a\&b$, such that a is an element of A and that the second component b is an element of the type B_a, indicated by the first element of the pair. We can interpret dependent product types $X : A\#B$ as existentially quantified propositions: there is an X of type A with property B. The proof of such an existential proposition consists in the intuitionistic framework of an element of the type A and a proof of the property B for this X, which we might consider as ordered pair. There is one selector construct *spread*, a generalisation of the usual projection operators: supposed $s = a\&b \; in \; (x : A\#B)$, then $spread(s, [u, v, t])$ is defined to be $t_{[a,b/u,v]}$, i.e. t with u and v substituted by the left and right component of s respectively. The left projection and the right projection operators could be described as $spread(s, [u, \tilde{}, u])$ and $spread(s, [\tilde{}, v, v])$, respectively.

2.13.1 Type Formation

1 ⊢(intro(X:A# ~),H==>u(I) *ext* X:A#B,
[==>A in u(I), [X:A]==>u(I) *ext* B]):-
syntax(H,A),free([X],H).

refinement rule: $X : A\#B$ is a possible refinement for any universe, if A is a type, and B is a refinement of the same universe under the assumption of the existence of some X of type A. This is a stepwise refinement rule. You make a partial decision (to use a dependent product type over A), but let the type of the second components be open. So there is a subgoal stating that your initial universe is inhabited, which would result (during further refinement) in some type B.

2 ⊢(intro(new[Y]),H==>(X:A#B) in u(I),
[==>A in u(I), [Y:A]==>By in u(I)]):-
free([Y],H),s(B,[Y],[X],By).

membership rule: $X : A\#B$ is a type of universe level i, i.e. a member of $u(i)$ if A is a type of universe level i, and $B_{[Y/X]}$ can be proven to be a type under the assumption of A being inhabited by an arbitrary element Y.

- ⊢(intro(new[Y]),_==>(X:A#B)=(XX:AA#BB) in u(I),
  [==>A=AA in u(I), [Y:A]==>By=BBy in u(I)]):-
  s(B,[Y],[X],By),s(BB,[Y],[XX],BBy).

  equality rule: Dependent product types are equal if the left base types are equal and the right base types can be proven equal under the assumption of the same free variable.

2.13.2 Constructors

5 ⊢(intro(*at*(I),F,new[Y]),H==>X:A#B *ext* F&G,
  [==>F in A, ==>Bf *ext* G, [Y:A]==>By in u(I)]):-
  syntax(H,F),free([Y],H),s(B,[F],[X],Bf),s(B,[Y],[X],By),level(I).
⊢(intro(*at*(I),F),H==>X:A#B *ext* F&G,
  [==>F in A, ==>Bf *ext* G, [X:A]==>B in u(I)]):-
  syntax(H,F),free([X],H),s(B,[F],[X],Bf),level(I).

refinement rule: If you supply a term F, the ordered pairs $F\&G$ may be regarded as a refinement of the dependent product type $X : A\#B$ at any universe level if F is a member of A, and G is a refinement of $B_{[F/X]}$, and if $B_{[Y/X]}$ can be proven to be a member of that universe under the assumption of Y being an arbitrary element of A. Proving the existence of some X of type A with the property B requires in the constructive framework always to supply such an element. The second rule describes the special case where X is a free variable in H, and may be used as a preferred variable identifier instead of Y. This has no influence on the logical interpretation, but allows you to exploit the intensional meaning of the variable identifier in further proof.

6 ⊢(intro(*at*(I),new[Y]),H==>(F&G) in (X:A#B),
  [==>F in A, ==>G in Bf, [Y:A]==>By in u(I)]):-
  free([Y],H),s(B,[F],[X],Bf),s(B,[Y],[X],By), level(I).
⊢(intro(*at*(I)),H==>(F&G) in (X:A#B),
  [==>F in A, ==>G in Bf, [X:A]==>B in u(I)]):-
  free([X],H),s(B,[F],[X],Bf),level(I).

membership rule: An ordered pair $F\&G$ is a member of the dependent product type $X : A\#B$ if F is a member of A and G is a member of $B_{[F/X]}$. and if $B_{[Y/X]}$ can be proven to be a member of that universe under the assumption of Y being an arbitrary element of A. The second rule describes the special case where X is a free variable in H, and may be used as a preferred variable

identifier instead of Y. This has no influence on the logical interpretation, but allows you to exploit the intensional meaning of the variable identifier in further proof.

- ⊢(intro,_==>U&V=UU&VV in (X:A#B),
  [==>U=UU in A, ==>V=VV in Bu]):-
  s(B,[U],[X],Bu).

  equality rule: Pairs are equal in the dependent product type, if the left components are equal in the base type, and the right components can be proven equal in the corresponding derived type.

2.13.3 Selectors

9 ⊢(elim(Z,new[U,V,N1]),H==>T *ext* spread(Z,[U,V,S]),
[[U:A, V:Bu, N1:Z=U&V in (X:A#B)]==>Tuv *ext* S]):-
decl(Z:X:A#B,H),
((free([X,V,N1], H),U=X,Bu=B);
(free([U,V,N1],H),s(B,[U],[X],Bu))
),
!,s(T,[U&V],[Z],Tuv).

refinement rule: The elimination of an arbitrary variable of a dependent product type in the hypothesis list, generates an *spread* term, the further refinement of which is determined by the subgoal of the *elim* step.

10 ⊢(intro(using(P:A#B),over(Z,T),new[U,V,W]),
H==>spread(E,[X,Y,S]) in Te,
[==>E in (P:A#B),
[U:A,V:Bu,W:E=U&V in (P:A#B)]==>Suv in Tuv]):-
τ_{var}(Z),syntax([Z:_|H],T),
τ_{var}(P),syntax(H,A),syntax([P:_|H],B),free([U,V,W],H),
s(B,[U],[P],Bu),s(S,[U,V],[X,Y],Suv),
s(T,[U&V],[Z],Tuv),s(T,[E],[Z],Te).

membership rule: A *spread term* is of a given type T_e if you can supply a type scheme $over(z, T_z)$, such that T_e is an instantiation of that type scheme for the base term E and the subterm of the spread term can be proven to be in the corresponding instantiation $T_{u,v}$, and if you can predict the type of the base term $using(p : A\#B)$. In most cases there is no direct depency between the type of the decision term and the current value of the base term of that decision term. To simplify the automatic proof of these wellformedness goals there is a *derived rule*, which assumes the result type to be constant. Still there is the problem of supplying the type information for the base term. For the simplest case, that the base term is a single variable, which is declared

in the hypothesis list, there is again a *derived rule* which simply accesses the declaration from the hypothesis list for generating the type of the base term.

- ⊢(intro(using(P:A#B),new[U,V,W]), H==>spread(E,[X,Y,S]) in T,
  [==>E in (P:A#B),
  [U:A,V:Bu,W:E=U&V in (P:A#B)]==>Suv in T]):-
  derived,syntax(H,A),syntax([P:_|H],B),τ_{var}(P),free([U,V,W],H),
  s(B,[U],[P],Bu),s(S,[U,V],[X,Y],Suv).
- ⊢(intro(new[U,V,W]), H==>spread(E,[X,Y,S]) in T,
  [==>E in (P:A#B),
  [U:A,V:Bu,W:E=U&V in (P:A#B)]==>Suv in T]):-
  derived,type(E,H,P:A#B), free([U,V,W],H),
  s(B,[U],[P],Bu),s(S,[U,V],[X,Y],Suv).

2.14 Quotient Types

Quotient types $A///[x, y, E]$ consist of elements T of the basic type A with equality given by the equivalence relation $E_{x,y}$.

2.14.1 Type Formation

1 ⊢(intro(A///[U,V,E],new[X,Y,Z,Wxy,Wyz]),H==>u(I) *ext* A///[U,V,E],
[==>A in u(I),
[X:A,Y:A]==>Exy in u(I),
[X:A]==>Exx,
[X:A,y:A,Wxy:Exy]==>Eyx,
[X:A,Y:A,Z:A,Wxy:Exy,Wyz:Eyz]==>Exz]):-
syntax(H,A),syntax(H,A///[U,V,E]),free([X,Y,Z,Wxy,Wyz],H),
s(E,[X,Y],[U,V],Exy),s(E,[Y,X],[U,V],Eyx),
s(E,[X,X],[U,V],Exx),s(E,[Y,Z],[U,V],Eyz),s(E,[X,Z],[U,V],Exz).

realisation rule: $A///[x, y, E]$ is a realisation for any universe, if A is a type over the same universe, E can be proven to be a type(proposition) under the assumption of x and y being elements of the type A, and if E is a equivalence relation, i.e. E fulfils the usual criteria for an equivalence relation.

2 ⊢(intro(new[X,Y,Z,Wxy,Wyz]),H==>(A///[U,V,E]) in u(I),
[==>A in u(I),
[X:A,Y:A]==>Exy in u(I),
[X:A]==>Exx,
[X:A,Y:A,Wxy:Exy]==>Eyx,
[X:A,Y:A,Z:A,Wxy:Exy,Wyz:Eyz]==>Exz]):-
free([X,Y,Z,Wxy,Wyz],H),

```
s(E,[X,Y],[U,V],Exy),s(E,[Y,X],[U,V],Eyx),
s(E,[X,X],[U,V],Exx),s(E,[Y,Z],[U,V],Eyz),s(E,[X,Z],[U,V],Exz).
```

membership rule: $A///[x, y, E]$ is a type of universe level i, i.e. a member of $u(i)$, under the same conditions as above.

```
6 ⊢(intro(new[R,S,T]),H==>(A///[X,Y,E])=(B///[U,V,F]) in u(I),
     [ ==> (A///[X,Y,E]) in u(I),
       ==> (B///[U,V,F]) in u(I),
       ==>A=B in u(I),
       [T:A=B in u(I),R:A,S:A]==>Ers=>Frs,
       [T:A=B in u(I),R:A,S:A]==>Frs=>Ers ]):-
     free([R,S,T],H),s(E,[R,S],[X,Y],Ers),s(F,[R,S],[U,V],Frs).
```

equality rule: Two quotient types are equal if the corresponding base types are equal and the defining relations can be proved to be equivalent.

2.14.2 Constructors

```
3 ⊢(intro(at(I)),_==>A///[X,Y,E] ext T,
     [ ==>A ext T, ==> (A///[X,Y,E]) in u(I) ]):-
     level(I).
```

refinement rule: Any refinement T of the base type of a quotient type may be used as a refinement of the quotient type at any universe level, if the quotient type is a member of that universe.

```
4 ⊢(intro(at(I)),_==>T in (A///[X,Y,E]),
     [ ==>T in A, ==> (A///[X,Y,E]) in u(I) ]):-
     noequal(T),level(I).
```

membership rule: T is an element of the quotient type at a given universe level if T is an element of the base type of that quotient type, and if the quotient type is a member of the given universe.

```
7 ⊢(intro(at(I)),_==>T=TT in (A///[X,Y,E]),
     [ ==> (A///[X,Y,E]) in u(I), ==>T in A, ==>TT in A, ==>Ett ]):-
     s(E,[T,TT],[X,Y],Ett),level(I).
```

equality rule: Two equivalence classes are equal, if their representatives are both members of the base type, and the equivalence relation can be proven between them.

2.14.3 Selectors

```
5 ⊢(elim(at(I),U,new[V,W,WE]),H==>S=SS in Tu,
     [ [V:A,W:A]==>Evw in u(I),
```

```
    ==>Tu in u(I),
    [V:A,W:A,WE:Evw]==>Sv=SSw in Tv ]):-
  decl(U:(A///[X,Y,E]),H), free([V,W,WE],H),
  s(E,[V,W],[X,Y],Evw),s(Tu,[V],[U],Tv),
  s(S,[V],[U],Sv),s(SS,[W],[U],SSw),
  level(I).
```

equality rule: If you want to prove an equality which depends on an element U of a quotient type, or more exactly, which depends on the fact that a quotient type is inhabited, than it is enough to prove the equality for U on both sides of the equality substituted by two arbitrary members of the equivalence class described by U.

- ⊢(elim(*at*(I),U,new[V,W,WE]),H==>S in Tu,
  ```
  [ [V:A,W:A]==>Evw in u(I),
    ==>Tu in u(I),
    [V:A,W:A,WE:Evw]==>SSv=SSw in Tv ]):-
  decl(U:(A///[X,Y,E]),H), free([V,W,WE],H),
  s(E,[V,W],[X,Y],Evw),s(Tu,[V],[U],Tv),
  s(S,[V],[U],SSv), s(S,[W],[U],SSw),
  level(I).
  ```

  equality rule: If you want to prove a membership goal which depends on an element U of a quotient type, or more exactly, which depends on the fact that a quotient type is inhabited, then it is enough to prove equality of two new terms, corresponding to terms related by the equivalence relation U.

- ⊢(elim(*at*(I),D,new[WE]),H==>G,
  ```
  [ ==> (A///[X,Y,E]) in u(I),
    [WE:Evw]==> G]):-
  decl(D:(V=W in (A///[X,Y,E])),H), free([WE],H),
  s(E,[V,W],[X,Y],Evw),
  level(I).
  ```

  equality rule: If you want to prove a goal which depends on an equality between elements of a quotient type, then the extra hypothesis that the elements are related by the equivalence relation can be added (provided that the quotient type is well-formed).

2.15 Subset Types

Set types project the mathematical idea of *sets* into the the type theoretic framework: A subset $\{X : A \backslash B\}$ should consist of exactly the elements X of A which fulfil the property B, the equality relation is the same as in the basic type of the set, all operations over the basic type are directly applicable over the set type. But this

approach contradicts basic assumptions of the intuitionism. Strictly speaking the elements of a subset $\{X : A\backslash B\}$ should be pairs $x\&b$, where x is an element of A and b is an element of B_x, i.e. a proof that x fulfils the defining property of the set. This concept of sets could easily be simulated using the dependent product types. But using this strict interpretation leads to a lot of problems in practical work. It would be impossible to use the operations which are defined on the basic type directly on the subset type. You can imagine the effort that would be necessary, to work in natural numbers, defined as a subset of the type *int*, if there were no access to the operations and rules available for integers. To overcome this dilemma, we opted for a compromise: The elements of the set type are really the elements the basic type, but they are connected with a hidden information, the extract term of the proof that these elements belong to the subset. If you refer to a hypothesis stating that a certain element belongs to a subset, you can use the fact that the defining property of the subset is fulfilled by that element. But as soon as you use this property in a constructive way, the extract term of your proof becomes tagged with an *assert*(...) term, describing the characteristic property of the set type. Such an *assert*(...) term is in general not executable, because it requires the proof of the property at run time. For certain simple properties this can be done. But this is not the point: the main idea is that these *assert*(...) terms are hidden in the extract term of the top level goal in most applications, because they appear in subgoals of a proof step which yields a constant extract term (for example *axiom*). That means you can use the subset types in the ordinary mathematical sense: all operations and functions defined on the basic type are directly available on the subset, but you should not use the defining property of a set type in constructive parts of the proof. If you are not sure about the effects, test the extract term of the top level goal.

2.15.1 Type Formation

```
1 ⊢(intro({X:A\ ~}),H==>u(I) ext {X:A\B},
      [ ==>A in u(I), [X:A]==>u(I) ext B ]):-
      syntax(H,A),free([X],H).
```

refinement rule: $\{X : A\backslash B\}$ is a possible refinement for any universe, if A is a type in that universe, and B is a refinement of the same universe under the assumption of X being an arbitrary element of A.

```
2 ⊢(intro(new[Y]),H==>{X:A\B} in u(I),
      [ ==>A in u(I), [Y:A]==>By in u(I) ]):-
      free([Y],H),s(B,[Y],[X],By).
  ⊢(intro,H==>{X:A\B} in u(I),
      [ ==>A in u(I), [X:A]==>B in u(I) ]):-
      free([X],H).
```

membership rule: $\{X : A\backslash B\}$ is a type of universe level i, i.e. a member of $u(i)$, if A is a type in $u(i)$ and $B[Y/X]$ can be proven to be a type in the same universe under the assumption of Y being an arbitrary element of A.

```
10 ⊢(intro(new[Z]),H==>{X:A\B}={Y:AA\BB} in u(I),
       [ ==>A=AA in u(I),
         [Z:A]==>Bz=>BBz,
         [Z:A]==>BBz=>Bz ]):-
       free([Z],H),s(B,[Z],[X],Bz),s(BB,[Z],[Y],BBz).
```

equality rule: Two set types are equal if the base types are equal and the defining relations can be proven to be equivalent over the base type.

2.15.2 Constructors

```
5 ⊢(intro(at(I),T,new[Y]),H==>{X:A\B} ext T,
       [ ==>T in A, ==>BB, [Y:A]==>By in u(I) ]):-
       syntax(H,T),free([Y],H), s(B,[T],[X],BB),s(B,[Y],[X],By),level(I).
  ⊢(intro(at(I),T),H==>{X:A\B} ext T,
       [ ==>T in A, ==>BB, [X:A]==>B in u(I) ]):-
       syntax(H,T),free([X],H), s(B,[T],[X],BB),level(I).
```

realisation rule: If you supply a term T, this term may be regarded as a realisation of the set type $\{X : A\backslash B\}$ at any universe level if T is a member of A and fulfils B, i.e. $B_{[T/X]}$ can be proven, and if $B_{[Y/X]}$ can be proven to be a member of that universe under the assumption of Y being an arbitrary element of A. Proving the existence of some X of type A with the property B requires in the constructive framework always the exhibition of a term yielding such an element, and the proof of the defining property of the subset. The second rule describes the special case where X is a free variable in H, and may be used as a preferred variable identifier instead of Y. This has no influence on the logical interpretation, but allows you to exploit the intensional meaning of the variable identifier in further proof.

```
6 ⊢(intro(at(I),new[Y]),H==>T in {X:A\B},
       [ ==>T in A, ==>BB, [Y:A]==>By in u(I) ]):-
       noequal(T),free([Y],H),s(B,[T],[X],BB),s(B,[Y],[X],By), level(I).
  ⊢(intro(at(I)),H==>T in {X:A\B},
       [ ==>T in A, ==>BB, [X:A]==>B in u(I) ]):-
       noequal(T),free([X],H),s(B,[T],[X],BB),level(I).
```

membership rule: T is a member of a set type, if T is a member of the base type and the defining property of the set type can be proven for T.

```
• ⊢(intro(at(I),new[Y]),H==>T=TT in {X:A\B},
```

[==>T=TT in A, ==>BB, [Y:A]==>By in u(I)]):-
free([Y],H),s(B,[T],[X],BB),s(B,[Y],[X],By), level(I).
⊢(intro(*at*(I)),H==>T=TT in {X:A\B},
[==>T=TT in A, ==>BB, [X:A]==>B in u(I)]):-
free([X],H),s(B,[T],[X],BB),level(I).

equality rule: Two elements of a set type are equal if they are equal in the base type.

2.15.3 Selectors

9 ⊢(elim(*at*(I),U,new[X,Y,Z]),H==>T *ext* σ(F,[U,assert(Bu)],[X,Y]),
[[X:A]==>Bx in u(I),
[X:A,Y:Bu,Z:X=U in A]==>Tu *ext* F]):-
decl(U:{V:A\B},H), free([X,Y,Z],H),
s(B,[X],[V],Bx),s(B,[U],[V],Bu),s(T,[X],[U],Tu),level(I).

refinement rule: This rule describes exactly the effect of exploiting the characteristic property of a set appearing in the hypothesis list. There is a derived rule, which handles the special case, that you can infer that an element X is an element of the base type of a set, if you know that X is an element of an subtype.

- ⊢(intro,H==>X in T, []):-
derived,member(X:{_:TT_},H), α(T,TT).

2.16 Recursive Types

Recursive types have the general form $rec(z, B_z)$, where z is a free variable in B_z. Invariably the form used is $rec(z, A\backslash B_z)$, where z does not appear in A. A is called the *base type* and B_z the *iterator* of the recursive type. The elements of such an recursive type are left injection terms built from the elements of A, and right injection terms built up from elements of the order $n>0$ of B_z according to the iterator. Elements of the order 1 of B_z are those elements which are built up assuming as basis elements of z only left injection terms from A. Elements of the order $n>1$ of B_z are those which are built up assuming as basis elements of z only left injection terms from A or right injection terms from elements of the order $n-1$ of B_z. Occurrences of the free variable are not allowed in the left hand side of function type, of in a functiom application, or in the first argument of an induction term.

Let us consider as an example the type $rec(z, int\backslash z\#z)$, which defines exactly the type of binary trees over the type *int*. Elements of $rec(z, int\backslash z\#z)$ are for example:

```
inl(i1),
```

```
inr(inl(i1) & inl(i2)),
inr(inr(inl(i1) & inl(i2)) & inl(i3)),
inr(inr(inl(i1) & inr(inl(i2) & inl(i3))) & inl(i4)),
```

where the i_j are elements of the base type *int*.

The construction of elements of a recursive type is described using the constructors for the base type $A\backslash B_z$. For recursive types there is a selector term $rec\_ind(x, [v, w, t])$ defined which describes a term of a type T as if there were a (recursive) mapping from the type $rec(z, A\backslash B_z)$ into T, defined by t, which decides whether the argument w is a left injection term coming from A or is constructed in some other way from elements of the recursive type again. In the latter case it is assumed that v already describes a mapping for the terms of lower order into T, which might be applied on the appropriate subterms extracted from w.

Let us consider again the type of binary trees over integers $rec(z, int\backslash z\#z)$. The *weight* of a tree, i.e. the sum of the integers in the tips of that tree could be defined by the induction term:

```
rec_ind(x,[v,w,decide(w,[i,i],
                  [p,spread(p,[l,r,v of l+v of r])])])
```

for $x = inl(i_1)$ this term would obviously yield i_1, for $x = inr(inl(i_1)\&inl(i_2))$ the variable p would become bound to $inl(i_1)\&inl(i_2)$ which results in l and r becoming $inl(i_1)$ and $inl(i_2)$ respectively. The recursive application of v yields then i_1 and i_2 which gives together the value $inl(i_1) + inl(i_2)$ for the induction term.

Let $rec(z, A\backslash B_z)$ be an arbitrary recursive type, then the type B' obtained from B_z by substituting the recursive type itself for z has exactly the same members as the recursive type. Therefore B' could be considered as another (unrolled) representation for the same (recursive) type.

2.16.1 Type Formation

```
1 ⊢(intro(new[Y]),H==>rec(Z,T) in u(I),
      [ [Y:u(I)]==>Ty in u(I) ]):-
      free([Y],H),s(T,[Y],[Z],Ty),
      \+ illegal_rec_type(rec(Z,T)).
```

membership rule: A recursive type is a member of a given universe, if its base type is a member of that universe, and if under the assumption of y being an arbitrary member of that universe the iterator applied on y can be proven to be a member of that universe too.

2.16.2 Constructors

```
2 ⊢(intro(at(I)),_==>rec(Z,T) ext X,
```

[==>TT *ext* X, ==>rec(Z,T) in u(I)]):-
s(T,[rec(Z,T)],[Z],TT), level(I).

refinement rule: A refinement for any recursive type can be constructed as a refinement of the unrolled recursive type.

3 ⊢(intro(*at*(I)),_==>X in rec(Z,T),
[==>X in TT, ==>rec(Z,T) in u(I)]):-
s(T,[rec(Z,T)],[Z],TT), level(I).

membership rule: A term X is a member of a recursive type, if X can be proven to be a member of the unrolled recursive type.

4 ⊢(unroll(X,new[Y,Z]),H==>G *ext* σ(F,[axiom,X],[Z,Y]),
[[Y:TT,Z:Y=X in TT]==>GG *ext* F]):-
decl(X:rec(V,T),H), free([Y,Z],H),
s(T,[rec(V,T)],[V],TT),s(G,[Y],[X],GG).

equality rule: For each variable of a recursive type one can derive an element of the unrolled type, which is equal to the original one. There is a derived rule, which allows the efficient handling of wellformedness goals generated in proofs concerning recursive types.

- ⊢(intro,H==>X in T0,[]):-
derived,decl(X:rec(V,T),H),s(T,[rec(V,T)],[V],TT),α(TT,T0).

2.16.3 Selectors

5 ⊢(elim(*at*(I),X,new[U,U0,V,W]),H==>G *ext* rec_ind(X,[V,W,F]),
[[U:u(I), U0:X:U=>(X in rec(Z,T)),
V:X:U=>G,
W:TT]==>Gw *ext* F]):-
decl(X:rec(Z,T),H), free([U,U0,V,W],H),
s(G,[W],[X],Gw),s(T,[U],[Z],TT), level(I).

refinement rule: Supposed U is an arbitrary subtype of the recursive type, (this property is described by U_0) and V is already declared as a mapping from that subtype into G, and is W an element of the next higher level of unrolling, i.e. an element of $A \backslash B_u$, and G can be refined to F then we can extend this structural induction step into a fully defined induction term.

6 ⊢(intro(*at*(I),using(rec(Z,T)),over(X,S),new[U,U0,V,W]),
H==>rec_ind(R,[G,J,K]) in Sr,
[[U:u(I), U0:W:U=>(W in rec(Z,T)), V:(W:U=>Su),W:Tu]==>KK in Su,
==>R in rec(Z,T)]):-
τ_{var}(X),syntax([X:_|H],S),τ_{var}(Z),syntax([Z:_|H],T),

free([U,U0,V,W],H), level(I), s(S,[R],[X],Sr),
s(S,[U],[X],Su), s(T,[U],[Z],Tu),s(K, [V,W], [G,J], KK).

membership rule: Suppose that U is an arbitrary subtype of the recursive type provided (property U_0), and V is a mapping into the corresponding type, supplied by the type scheme $over(x, S)$, and W is an element of the next higher level of unrolling. If one can prove under these assumptions that the base term really belongs to the recursive type supplied, then the value of the structural induction term belongs to the appropriate instantiation of S. Again there are derived rules handling the most usual cases of a constant type scheme and a directly accessible declaration for the base term.

- ⊢(intro(*at*(I),using(rec(Z,T)),new[U,U0,V,W]),
  H==>rec_ind(R,[G,J,K]) in S,
  [[U:u(I), U0:W:U=>(W in rec(Z,T)), V:(W:U=>S),W:Tu]==>KK in S,
  ==>R in rec(Z,T)]):-
  derived,syntax([Z:_|H],T),τ_{var}(Z),free([U,U0,V,W],H), level(I),
  s(T,[U],[Z],Tu),s(K, [V,W], [G,J], KK).
- ⊢(intro(*at*(I),new[U,U0,V,W]),H==>rec_ind(R,[G,J,K]) in S,
  [[U:u(I), U0:W:U=>(W in rec(Z,T)), V:(W:U=>S),W:Tu]==>KK in S,
  ==>R in rec(Z,T)]):-
  derived,type(R,H,rec(Z,T)), free([U,U0,V,W],H), level(I),
  s(T,[U],[Z],Tu),s(K,[V,W],[G,J], KK).

2.17 Acc Types

Acc types have the general form $acc(A, R)$. A, the base type, is a set well-ordered by the relation R. For *acc* types there is a selector term $wo\_ind(X, [W, U, Ts])$. Here X is the recursion argument. Let us suppose that (a) $W : A$ and (b) that if U is a member of the dependent function type mapping members V of the subset $\{V : A \backslash R\ of\ V\ of\ W\}$ onto members of a type $T(V)$ depending on V, then T is a member of $T(W)$. Then $wo\_ind(X, [W, U, Ts])$ is a member of $T(X)$.

2.17.1 Type Formation

- ⊢(intro(acc(A,~)),H==>u(I) *ext* acc(A,R),
  [==>A in u(I), ==>(A=>A=>u(I)) *ext* R]):-
  syntax(H,A).

  refinement rule: $acc(A, R)$ is a possible refinement for any universe, if A is a type in that universe, and R is a refinement of $A => A => u(I)$.

- ⊢(intro,_==>acc(A,R) in u(I),

[==>A in u(I),==>R in (A=>A=>u(I))]).

membership rule: $acc(A, R)$ is a type of universe level,i.e. a member of $u(I)$, if A is a type in that universe, and R is a member of $A => A => u(I)$.

- ⊢(intro(new[X,Y]),H==>acc(A,R)=acc(Ax,Rx) in u(I),
  [==>A=Ax in u(I),
  [X:A,Y:A]==>R of X of Y => Rx of X of Y,
  [X:A,Y:A]==>Rx of X of Y => R of X of Y]):-
  free([X,Y],H).

  equality rule: Two *acc* types are equal if the base types are equal and the defining relations can be proved to be equivalent over the base type.

2.17.2 Constructors

- ⊢(intro(*at*(I),X,new[U,V]),H==>acc(A,R) *ext* X,
  [==>X in A,[V:A,U:R of V of X]==>V in acc(A,R),
  ==>acc(A,R) in u(I)]):-
  syntax(H,X),free([U,V],H),level(I).

  realisation rule: If you supply a term X, this term may be regarded as a realisation of the acc type $acc(A, R)$ at any universe level if X is a member of A, and if V can be proved to be a member of $acc(A, R)$ on the assumption that V is a member of A and $R\ of\ V\ of X$, and that $acc(A, R)$ is a member of universe $u(I)$.

- ⊢(intro(*at*(I),acc,new[U,V]),H==>X in acc(A,R),
  [==>X in A,[V:A,U:R of V of X]==>V in acc(A,R),
  ==>acc(A,R) in u(I)]):-
  free([U,V],H),level(I).

  membership rule: X is a member of the the *acc* type $acc(A, R)$ if X is a member of A, and if V can be proved to be a member of $acc(A, R)$ on the assumption that V is a member of A and $R\ of\ V\ of\ X$, and that $acc(A, R)$ is a member of universe $u(I)$.

- ⊢(intro(*at*(I)),_==>T=Tx in acc(A,R),
  [==>T=Tx in A,==>R in (A=>A=>u(I))]):-
  level(I).

  equality rule: Two elements of an *acc* type are equal if they are equal on the base type.

2.17.3 Selectors

- ⊢(elim(X,wo,new[U,V,W]),H==>T *ext* wo_ind(X,[W,U,Ts]),

```
[[W:A,U:(V:{V:A\R of V of W}=>Tsv)] ==> Tsw ext Ts]):-
decl(X:acc(A,R),H),free([U,V,W],H),
s(T, [V], [X], Tsv),
s(T, [W], [X], Tsw).
```

refinement rule: The elimination of an arbitrary variable of type $acc(A, R)$ in the hypothesis list, generates an induction term, the further refinement of which is determined by the subgoals of the elim step.

- ```
  ⊢(intro(using(acc(A,R)),over(Z,T),new[U,V,W]),
      H==>wo_ind(E,[X,Y,Ts]) in Te,
      [==>E in acc(A,R),
      [V:(W:{U:A\R of U of E}=>Tsw)]==>Tsev in Te]):-
      τvar(Z),syntax([Z:_|H],T),free([U,V,W],H),
      s(T,[E],[Z],Te),
      s(Ts,[E,V],[X,Y],Tsev),
      s(T,[W],[Z], Tsw).
  ```

  membership rule: An induction term is of a given type Te if you can supply a type schema $over(Z, T)$, such that Te is an instantiation of that type schema for E and the subterm of the induction term can be proved to be in the corresponding Ts. To simplify the automatic proof of these well-formedness goals there is a derived rule, which asssumes the result type to be constant.

  - ```
    ⊢(intro(using(acc(A,R)),new[U,V,W]),H==>wo_ind(E,[X,Y,Ts]) in T,
        [==>E in acc(A,R),
        [V:(W:{U:A\R of U of E}=>T)]==>Tsev in T]):-
        derived,free([U,V,W],H),
        s(Ts,[E,V],[X,Y],Tsev).
    ```

- ```
  ⊢(reduce(using(acc(A,R))),H==>wo_ind(E,[X,Y,T])=Tsev in _,
      [==>E in acc(A,R)]):-
      free([R],H),
      s(T,[E,lambda(R,wo_ind(R,[X,Y,T]))],[X,Y],Tsev).
  ```

  equality rule: This rule describes the value of the induction term under certain assumptions.

2.18 Membership and Equality

Membership and *equality* are type schemes which have been introduced to keep the theoretical system closed. For each term A and T, $A\ in\ T$ is a type if T is a type and A is a member of T. Analogous $A = B\ in\ T$ is a type for any type T and arbitrary elements A and B of T. The membership and equality terms are inhabited by axiom,
  ```

if there is a proof for membership or equality. In this section only the *catch all* rules for membership and equality are given. The domain specific rules, which exploit the particular structure of the subterms, are given in the sections corresponding to the type of those subterms.

2.18.1 Type Formation

```
1 ⊢(intro(~ in A),H==>u(I) ext (X in A),
      [ ==>A in u(I), ==>A ext X ]):-
      syntax(H,A).
```

refinement rule: Any membership term over an arbitrary type A is a type.

```
2 ⊢(intro,_==>(X in A) in u(I),
      [ ==>A in u(I), ==>X in A ]):-
      noequal(X).
```

membership rule: The membership term X *in* A is a type if A is a type and X is a member of A.

```
1 ⊢(intro(~ = ~ in A),H==>u(I) ext (X=Y in A),
      [ ==>A in u(I), ==>A ext X, ==>A ext Y ]):-
      syntax(H,A).
```

refinement rule: Any equality term over an arbitrary type A is a type.

```
2 ⊢(intro,_==>(X=Y in A) in u(I),
      [ ==>A in u(I), ==>X in A, ==>Y in A]):-!.
```

membership rule: The equality term $X = Y$ *in* A is a type, if A is a type and X and Y are members of A.

2.18.2 Constructors

```
4 ⊢(intro,H==>X in T,[]):-
      noequal(X),decl(X:TT,H),α(TT,T).
```

realisation rule: If a variable X is declared to be of the type T then it is a member of T, and the membership term refines to *axiom*.

```
5 ⊢(intro(at(I)),_==>X=XX in T, [ ==>T in u(I), ==>X in T ]):-
      α(X,XX),level(I),\+((T=u(J),I=<J)).
```

realisation rule: The equality term $X = X'$ *in* T refines to axiom, if the two terms X and X' are syntactically equivalent, i.e. if they are identical except for possible renaming of bound variables, and if T is a type and X and/or X' is a member of T.

3 ⊢(intro,_==>axiom in (X in A), [==>X in A]):- noequal(X).

membership rule: The membership type X *in* A is inhabited by axiom, if X is a member of A.

3 ⊢(intro,_==>axiom in (X=Y in A), [==>X=Y in A]).

membership rule: The equality type $X = Y$ *in* A is inhabited by axiom, if X is equal to Y in A.

2.19 Universes

2.19.1 Type Formation

1 ⊢(intro(u(I)),_==>u(J) *ext* u(I),[]):- level(I),I<J.

realisation rule: Lower universes are realisations for higher universes.

2 ⊢(intro,_==>u(I) in u(J),[]):- I<J.

membership rule: A lower universe is a member of a higher universe.

2.19.2 Constructors

- **refinement rules** and **membership rules:** The type formation rules in the preceding chapters may be considered as refinement and membership rules for constructors in $u(i)$.

6 ⊢(intro(u(I)),_==>T in u(J), [==>T in u(I)]):-
level(I),I<J.

membership rule: This rule describes the *cumulativity* of the chain of universes. There is a derived rule which handels the case of a single variable being an element of a universe.

  - ⊢(intro,H==>X in u(I), []):-
derived, decl(X: u(J),H), J<I.

2.20 Miscellaneous

This section contains rules which are applicable in most situations, so it makes no sense to return the corresponding rule specifications in response to an enquiry of the form $apply(X)$.

0 ⊢(X,_,_):-var(X),!,fail.

2.20.1 hypothesis

```
1 ⊢(hyp(V),H==>V in T,[]):-
        atom(V),
        !,
        decl(V:TT,H),
        α(T,TT).
  ⊢(hyp(X),H==>A in T ext X,[]):-
        \+ A = (_=_),
        !,
        decl(X:HT,H),
        ( HT = (AA in TT); HT = (AA=_ in TT); HT =(_=AA in TT)),
        α(T,TT),
        α(A,AA).
  ⊢(hyp(X),H==>A ext X,[]):-
     decl(X:AA,H),α(A,AA).
```

This rules enforces the direct application of a hypothesis for proving the current goal. In the standard mode of operation the hypothesis list is checked before any attempt to prove a goal. But this test is switched off in the *pure* mode. Therefore you can use the *hyp* rule. Furthermore this rule simplifies the proof of membership goals if you have a suitable equality in the hypothesis list.

2.20.2 sequence

```
2 ⊢(seq(T,new[X]),H==>G ext lambda(X,E)of F,
        [ ==>T ext F, [X:T]==>G ext E ]):-
        syntax(H,T),free([X],H).
```

The *seq*-rule provides some element of *bottom up programming* or *forward chaining* to the logic, which allows a better global structuring of the proof tree and the synthesised programs.

2.20.3 thinning

```
3 ⊢(thin(ToThin), H==>G ext E,
        [ (-ThinL) ==> G ext E ]):-
        freevarsinterm( G, NotThin ),
        extend_thin( ToThin, H, NotThin, ThinL ).
```

2.20.4 lemma

```
4 ⊢(lemma(T,new[X]),H==>G ext σ(E,[term_of(T)],[X]),
        [ [X:C]==>G ext E ]):-
```

```
    free([X],H),!,ϑ_theorem(T) =: P, ϑ_thm =:CT, functor(CT,N,_), \+ N=T,
    (status_0(P,complete); status_0(P,complete(because))),
    P = π(_==>C,_,_,_).
⊢(intro,_==>term_of(T) in Type, [] ) :-
    ϑ_theorem(T) =: P,
    P = π([]==>TT,_,_,_),
    (status_0(P,complete); status_0(P,complete(because))),
    α(TT,Type).
/*
* veto-ed by Frankh, Andrew, and AlanS:
* ⊢(intro,_==>({Name} in Type), [ ==>(Body in Type) ]):-
* atom(Name),
* !,
* cdef(Name) =: ({Name}<==>Body).
* ⊢(intro,_==>(Tm in Type), [ ==>(ETm in Type) ]):-
* Tm =.. [Name|Args],
* cdef(Name) =: (Head<==>Body),
* !,
* Head =.. [Name|P],
* s(Body,Args,P,ETm).
*/
```

The *lemma*-rule gives access to external theorems, which have to be declared in the hypothesis list on the toplevel of this proof.

The converse *intro* rule for extract term formation allows the type of an extract term to be deduced from the theorem it was extracted from.

2.20.5 def

```
5 ⊢(def(T,new[X]),H==>G ext E,
    [ [X:term_of(T)=Ex in C]==>G ext E ]):-
    free([X],H), !, ϑ_theorem(T) =: P,status_0(P,complete),
    P=π(_ ==> C,_,_,_),
    extract(P,Exx),polish(Exx,Ex).
```

The *def*-rule explicitly provides access to the extract term of a theorem declared in the global hypothesis list.

2.20.6 explicit intro

```
5 ⊢(intro(explicit(X)),H==>T ext X, [ ==>X in T ]):-syntax(H,X).
```

realisation rule: This rule allows to supply the extract term for the current goal explicitly. In many situations you are able to supply this term directly

because of your programming or theorem proving experience. Furthermore this rule allows you to simplify the proof structure by extracting a term from an(other) proof tree and supplying this term directly as extract term.

2.20.7 substitution

```
7 ⊢(subst(at(I),over(Z,T),T1=T2 in T0),H==>TT1 ext E,
      [ ==>T1=T2 in T0, ==>TT2 ext E, [Z:T0]==>T in u(I) ]):-
      free([Z],H),s(T,[T1],[Z],TT),α(TT,TT1),s(T,[T2],[Z],TT2),
      syntax(H,T2 in T0),level(I).
  ⊢(subst(at(I),hyp(N),over(Z,T),T1=T2 in T0),H==>G ext E,
      [ ==>T1=T2 in T0, +([N:TT2])==>G ext E, [Z:T0]==>T in u(I) ]):-
      free([Z],H),decl(N:TT1,H),
      s(T,[T1],[Z],TT),α(TT,TT1),s(T,[T2],[Z],TT2),
      syntax(H,T2 in T0),level(I).
```

The *substitution*-rule allows the partial substitution of multiple occurences of a subterm by another term.

2.20.8 direct computation

```
8 ⊢(compute(using(X)),_==>G ext E, [ ==>GG ext E ]):-
      correct_tag([],G,X),
      compute_using(X,GG),
      \+ X = GG.
  ⊢(compute(using(X),not(I)),_==>G ext E, [ ==>GG ext E ]):-
      correct_tag(I,G,X),
      compute_using(X,GG),
      \+ X = GG.
  ⊢(compute(hyp(N),using(X)),H==>G ext E,
          [ +([N:TT]) ==>G ext E ]):-
      decl(N:T,H),
      correct_tag([],T,X),
      compute_using(X,TT),
      \+ X = TT.
  ⊢(compute(hyp(N),using(X),not(I)),H==>G ext E,
          [ +([N:TT]) ==>G ext E ]):-
      decl(N:T,H),
      correct_tag(I,T,X),
      compute_using(X,TT),
      \+ X = TT.

  ⊢(compute(X),_==>G ext E, [ ==>GG ext E ]):-
      compute_old(X,G,GG).
```

```
⊢(compute(hyp(N),X),H==>G ext E,
        [+([N:TT])==>G ext E ]):-
        decl(N:T,H), compute_old(X,T,TT).

9 ⊢(normalise,_==>G ext E, [ ==>GG ext E ]):-
      normalise([],G,GG),
      \+ G = GG.
  ⊢(normalise(not(I)),_==>G ext E, [ ==>GG ext E ]):-
      normalise(I,G,GG),
      \+ G = GG.
  ⊢(normalise(hyp(N)),H==>G ext E,
            [ +([N:TT]) ==>G ext E ]):-
      decl(N:T,H),
      normalise([], T, TT ),
      \+ T = TT.
  ⊢(normalise(hyp(N),not(I)),H==>G ext E,
            [ +([N:TT]) ==>G ext E ]):-
      decl(N:T,H),
      normalise(I, T, TT ),
      \+ T = TT.
```

The *compute*-rule allows the parallel manipulation of different subterms by means of rewriting operations. The subterms are specified using tag frames. A *tag frame* is a term with the same structure as the original term, where several subterms are replaced by tags of the form [[...]]. A *tag* describes the operations which have to be performed on the original subterm of the same position. There are the following operations available:

- multiple application of rewrite rules $[[n]]$ (n-times, if $n>0$, or iterative as far as possible for $n = 0$);
- *folding* and *unfolding* of definitions specified by tagged terms of the form $[[fold(d)]]$ and $[[unfold(d)]]$ or $[[unfold]]$ respectively;
- *simplification*, i.e. substitution of the subterm with the result of the evaluation of the subterm in the form $[[simplify]]$; and
- *equality rule application* in the form $[[R]]$, where R is the specification of a rule applyable to a goal of the form $t = t' in T$ where t matches with the original subterm and t' gives the rewriting or vice versa in the case of inverse rule application $[[inv(R)]]$.

2.20.9 equality and arith

equality is supplied as elementary inference rule as in the Nuprl system. The similar *arith* rule is not implemented. Nevertheless both should be better treated via tactics with the same functionality as soon as possible.

10 ⊢(equality,H==>A=B in T, [==>A in T, ==>B in T]):-
decide$_=$(A=B in T,H).

11 ⊢(arith,H==>G *ext* E,S) :- fail.

2.21 Shorthands

- ⊢(reduce(left,D),H==>X=Y in T, [==>XX=Y in T | S]):-
  ⊢(reduce(D),H==>X=XX in T, S).
  ⊢(reduce(right,D),H==>X=Y in T, [==>X=YY in T | S]):-
  ⊢(reduce(D),H==>Y=YY in T, S).

  These rules allow a partial reduction of one side of an equality. One could get the same effect using the more general compute rule.

- ⊢(simplify,P,S):-⊢(compute([[simplify]]),P,S).

  A simplified user interface for the simplification of terms.

- These final rules extend the simple rules dealing with introduction,reducing goals of the form $t_1 = t_2 in T$ to those of the form: $t in T$.

```
⊢(intro,H==>(T1 = T2 in Type), SubGoals ) :-
    functor(T1,N,_),
    functor(T2,N,_),
    ⊢(intro,H==>T1 in Type, SubGoals1),
    ⊢(intro,H==>T2 in Type, SubGoals2),
    equal_combine( SubGoals1, SubGoals2, SubGoals).
⊢(intro(A),H==>(T1 = T2 in Type), SubGoals ) :-
    functor(T1,N,_),
    functor(T2,N,_),
    ⊢(intro(A),H==>T1 in Type, SubGoals1),
    ⊢(intro(A),H==>T2 in Type, SubGoals2),
    equal_combine( SubGoals1, SubGoals2, SubGoals).
⊢(intro(A,B),H==>(T1 = T2 in Type), SubGoals ) :-
    functor(T1,N,_),
    functor(T2,N,_),
    ⊢(intro(A,B),H==>T1 in Type, SubGoals1),
    ⊢(intro(A,B),H==>T2 in Type, SubGoals2),
```

```
        equal_combine( SubGoals1, SubGoals2, SubGoals).
    ⊢(intro(A,B,C),H==>(T1 = T2 in Type), SubGoals ) :-
        functor(T1,N,_),
        functor(T2,N,_),
        ⊢(intro(A,B,C),H==>T1 in Type, SubGoals1),
        ⊢(intro(A,B,C),H==>T2 in Type, SubGoals2),
        equal_combine( SubGoals1, SubGoals2, SubGoals).
```

- ```
    ⊢(R,P,S):-
        R=..[F|T], \+ append([at(_)],_,T),
        member(F,[intro,elim,subst]),
        ϑ_universe=:I,RR=..[F,at(I)|T],
        ⊢(RR,P,S).
    ⊢(R,P,S):-
        R=..[F|T], \+ append(_,[new(_)],T),
        member(F,[intro,elim,decide,equality,unroll,compute,seq,lemma,def]),
        append(T,[new(_)],TT),RR=..[F|TT],
        ⊢(RR,P,S).
    ```

  The last two rules describe the automatic generation of $at(..)$ and $new[...]$ parameters in rule specifications.

- ⊢(because,_==>_ *ext* atom(incomplete),[]).

  - a blatant cheat.

```
equal_combine( [],[],[]).
equal_combine( [==>T1 in T|Rest1], [==>T2 in T|Rest2], [==> T1 in T|RestC] ) :-
    α( T1, T2 ),
    !,
    equal_combine( Rest1, Rest2, RestC ).
equal_combine( [==>T1 in T|Rest1], [==>T2 in T|Rest2], [==> T1=T2 in T|RestC] ) :-
    equal_combine( Rest1, Rest2, RestC ).
equal_combine( [[H1|T1]==>G1|Rest1], [[H2|T2]==>G2|Rest2],[[H1|TT]==>GG|RestC])
  :- α( H1,H2 ), !,
    equal_combine( [T1==>G1|Rest1],[T2==>G2|Rest2], [TT==>GG|RestC] ).
/* we can't simply drop non-convertible declarations!
equal_combine( [[_|T1]==>G1|Rest1], [[_|T2]==>G2|Rest2],[TT==>GG|RestC])
  :- equal_combine( [T1==>G1|Rest1],[T2==>G2|Rest2], [TT==>GG|RestC] ).
*/
equal_combine( [[]==>G1|Rest1], [[]==>G2|Rest2],[[]==>GG|RestC])
  :- equal_combine( [==>G1|Rest1],[==>G2|Rest2], [==>GG|RestC] ).
```
```

Chapter 3

Tactics

3.1 Mark and Copy

$mark(file)$ and $copy(file)$ allow copying of arbitrary parts of the proof tree. $mark$ and *copy* might be considered as examples of built in tactics, thei do not use any hidden predicate. This example should encourage you in writing your own tactics more suitable for your applications. $mark$ writes to the file specified in its argument a sequence of Prolog clauses of the form $?-apply(t)$, $?-up$, and $?-down(i)$, which allows later the straightforward reconstruction of the proof tree below the current focus. The $mark$ file may be executed directly from $copy(file)$ as well as from the standard $consult(file)$. $mark$ and *copy* are a simple way of copying proof structures from one proof to another, as well as a tool for reproving (parts) of the current proof, and saving intermediate states of a subproof, when you decide to try another way of proving. Due to the fact, that the proof is saved in a quite readable ASCII file, you have the opportunity of modifying your proof tree using your favourite text editor.

```
mark(X):-atom(X), tell(X), markf(0), told,!.


copy(X):-consult(X).


markf(T):-refinement(R),
          ( var(R);
            writeclause(T,(apply(R)->true;abort)),
              TT is T+2, down(N),refinement(Q), \+ var(Q),
              writeclause(TT,down(N)),markf(TT),writeclause(TT,up),fail;
            !
          ),!.
```

```
writeclause(N,X):-tab(N),write_term((?-X),[quoted(true)]),write('.'),nl.
```

3.2 Equality and Arithmetic

One of the open research questions is the proper reorganization of the *arith* and *equality* rules as tactics. We have all preconditions fulfilled by extending the rule base by the appropriate primitive inference rules. To get an efficient working version of the system, both rules are still hard wired (as in the original Nuprl system). This section gives the source code for these inference rules, which you only should consider as an initial step.

3.2.1 Decision Procedure for Equalities

The decision procedure for equalities $A = B$ *in* T considers the reflexivity, symmetry and transitivity of the equality relation under T and all types which are convertible with T. The decision procedure works in three steps: first all equalities over types T' which are convertible to T are collected into a list, and then the list L of all elements which are equal to A under this restricted set of equalities is computed and then is checked whether B is convertible to an element of this list.

```
decide_=(A=B in T,H):-
    χ_=(T,H,H0), allequals([A],H0,L),!,member(BB,L),α(BB,B).


allequals(L1,T,L2):-equals(L1,T,L),(L=L1,L2=L; allequals(L,T,L2)),!.


equals(A,[],A).
equals(A,[B=C in _|T],L):-
    member(B,A),\+ member(C,A),!,equals([C|A],T,L).
equals(A,[B=C in _|T],L):-
    member(C,A),\+ member(B,A),!,equals([B|A],T,L).
equals(A,[_|T],L):-equals(A,T,L).


χ_=(_,[],[]).
χ_=(T,[_:A=B in TT|H],[A=B in TT|HH]):-α(T,TT),χ_=(T,H,HH).
χ_=(T,[A#B|H],HH):-χ_=(T,[A,B|H],HH).
χ_=(T,[_|H],HH):-χ_=(T,H,HH).
```

3.2.2 Decision Procedure for Arithmetic Properties

3.3 Syntax Checker for Type Theoretical Terms

The $syntax(H, X)$ predicate may be used to check the syntactic structure of a type theoretic term with respect to the declarations available in the hypothesis list H. It does not do any parsing, but simply checks the structure of a Prolog term. So, errors which appear already on the Prolog level may cause syntax error in the Prolog read routine. In the case of syntactic errors, $syntax(H, X)$ fails and generates error messages on standard output, otherwise it succeeds silently. τ_{var}(X) checks whether the parameter is a proper type theoretic variable. A type theoretic variable consists of a sequence of letters, digits and underscores beginning with a small letter, or a sequence of the characters "$+ - *\/\hat{}<>=`\sim:.?@\#\$\&$".

```
syntax(H==>G):-syntax0([],H==>G).
syntax(_,X):-var(X),!, direction=:backwards, !.
syntax(_,X):-var(X),!, direction=:forwards, !, fail.
syntax(_,new [_|_]):-!,fail.
syntax(_,at(_)):-!,fail.
syntax(H,over(Z,T)):-τ_var(Z),syntax([Z:_|H],T).
syntax(H,X):-errors:=0,τ(H,X),errors=:I,!,I=:=0.
syntax(H,V,X):-errors:=0,τ'(H,V,X),errors=:I,!,I=:=0.


syntax0(H,[]==>G):-syntax(H,G).
syntax0(H,[D<==>P|HH]==>G):-
        D=..[F|A],free([F],H),τ'(H,A,P),syntax0([D<==>P|H],HH==>G).
syntax0(H,[T:(HL==>G ext E)|HH]==>GG):-
        free([T],H),τ(H,G),τ(H,E),syntax0([T:(HL==>G ext E)|H],HH==>GG).
syntax0(H,[T:(HL==>G)|HH]==>GG):-
        free([T],H),τ(H,G),syntax0([T:(HL==>G)|H],HH==>GG).


τ(H,X):-var(X),!,τ(H,'prolog variable not allowed').
τ(_,u(I)):-level(I),!.
τ(_,I):-integer(I),!.
τ(_,atom(X)):-atom(X),!.
τ(_,~):-fail.
τ(H,X):-τ_var(X),decl(X:_,H).

τ(_,X):- atomic(X), member(X,[atom,void,pnat,int,axiom,nil,unary,unit]),!.
```

```
τ(H,X):- X=..[F,A], member(F,[any,s,inl,inr,list,-]),τ(H,A),!.
τ(H,X):- X=..[F,A,B], member(F,[<,<*,&,::,#,=>,\,of,+,-,*,/,mod]),
        τ(H,A),τ(H,B),!.
τ(H,X):-
     X=..[F,A,B,S,T],
     member(F,[atom_eq,int_eq,pnat_eq,less,pless]),
     τ(H,A),τ(H,B),τ(H,S),τ(H,T),!.

τ(H,C:A#B):-τ(H,A),τ'(H,[C],B),!.
τ(H,C:A=>B):-τ(H,A),τ'(H,[C],B),!.
τ(H,A///[U,V,B]):-τ(H,A),τ'(H,[U,V],B),!.
τ(H,{C:A\B}):-τ(H,A),τ'(H,[C],B),!.
τ(H,{A\B}):-τ(H,A\B),!.
τ(H,lambda(A,B)):-τ'(H,[A],B),!.
τ(H,rec(A,T)):-τ'(H,[A],T),!.
τ(H,A=B in T):-τ(H,A),τ(H,B),τ(H,T),!.
τ(H,A in T):-τ(H,A),τ(H,T),!.
τ(H,acc(A,B)):-τ(H,A),τ(H,B).

τ(H,spread(A,[U,V,W])):-τ(H,A),τ'(H,[U,V],W),!.
τ(H,decide(A,[U,S],[V,T])):-τ(H,A),τ'(H,[U],S),τ'(H,[V],T),!.
τ(H,p_ind(A,B,[U,V,T])):-τ(H,A),τ(H,B),τ'(H,[U,V],T),!.
τ(H,cv_ind(A,[U,V,T])):-τ(H,A),τ'(H,[U,V],T),!.
τ(H,ind(A,[P,Q,S],B,[U,V,T])):-τ(H,A), τ'(H,[P,Q],S), τ(H,B), τ'(H,[U,V],T),!.
τ(H,list_ind(A,B,[U,V,W,T])):-
     τ(H,A),τ(H,B), τ'(H,[U,V,W],T),!.
τ(H,rec_ind(A,[U,V,T])):- τ(H,A),τ'(H,[U,V],T),!.
τ(H,wo_ind(A,[U,V,T])):- τ(H,A),τ'(H,[U,V],T),!.
τ(_,term_of(T)):- ϑ_theorem(T) =: _.
τ(H,X):-X=..[F|A],\+ F={},cdef(F) =: (XX<==>_),XX=..[F|AA],τ(H,A,AA),!.
τ(_,{Name}):- atom(Name),cdef(Name) =: ({Name}<==>_),!.
τ(H,Unknown) :- % try to load from a library
     autoload_defs_ass2(yes),
     once((Unknown = {F}; % function constants only
Unknown = term_of(F); % synths
Unknown =.. [F|[_|_]],
\+ oyster_functor(F))),
     \+ lib_present(def(F)), % circularity is possible
     (autoloading_def(F) -> (nl,write('Looping in autoload: aborting!'),nl,fail);
     true),
     assert(autoloading_def(F)),
     lib_load(def(F)),
     writef(' [ Autoloaded def(%t) ]\n',[F]),
```

```
    τ(H,Unknown), % start again.
    retractall(autoloading_def(F)).

τ(_,X):- nl,write('syntax error: '),write(X), errors=:I,J is I+1,errors:=J.
τ(_,[],[]).
τ(H,[A|T],[_|TT]):-τ(H,A),τ(H,T,TT).


τ'(H,V,A) :- remove(V,~,VV), is_set(VV), τ''(H,VV,A). % same as check_set?


τ''(H,[],A):-τ(H,A).
τ''(H,[U|T],A):-τ_var(U),τ''([U:dummy|H],T,A).

remove([],_,[]).
remove([X|T],X,D) :- remove(T,X,D).
remove([H|T],X,[H|D]) :- remove(T,X,D).


τ_var(X):-
    atom(X),
    oyster_functors( OF ),
    member(X, OF),
    !, fail.
τ_var(X):- atom(X).

oyster_functor(F) :-
    oyster_functors(Fs),
    member(F,Fs).
oyster_functors( [ atom, void, pnat, int, axiom, nil, ext, ==>, <==>, :, =>, #,
            \, ///, ~>, in, =, <, <=, +, -, *, /, mod, &, ::, of, list,
            lambda, rec, spread, decide, p_ind, ind, list_ind, rec_ind,
            acc, wo_ind, cv_ind, inr, inl,

  s,u,

            unit, unary, term_of, '.'] ).


fairly_nonalphanum([]).
fairly_nonalphanum([X|Y]):-
```

```
    member(X,[43,45,42,92,47,94,60,62,61,96,126,58,46,63,64,35,36,38]),
    fairly_nonalphanum(Y).


idtail([]).
idtail([H|T]):-(small_letter(H);capital_letter(H);digit(H);underscore(H)),idtail(T).
```

3.4 Pretty Printer for Type Theoretic Terms

$writeterm(T)$ and $writeterm(n,T)$ are the standard output predicates for type theoretic terms, where n stands for any initial indentation. $writeterm0(T)$ and $writeterm0(n,T)$ respectively write a compressed version of the term. Pretty printing requires an estimation of the string length of the output. To avoid multiple computation of that string length, the output term is first attributed with length information using the $weight$ predicate. $weight(X,Y)$ yields a tree Y with the same structure as X, but attributed with the estimated print length of X, as for example in $weight(a+b, w(w(a,1)+w(bb,2),6))$. This process of computing the output size is quite expensive, and should be subject to further optimisation. $fringe(n,X,XX)$ yields a term of limited depth n, the top level nodes of which are equivalent to X, omitted subterms are replaced by '...' .

```
writeterm(N,X):-vars:=0,copy(X,Y),weight(Y,W),prepare(Y,Z),ω(N,Z,W),nl.
writeterm(X):-writeterm(0,X).


writeterm0(N,X):-ϑ_fringe=:0, writeterm(N,X).
writeterm0(N,X):-ϑ_fringe=:F, fringe(F,X,Y),writeterm(N,Y).
writeterm0(X):-writeterm0(0,X).


ω(S,X,w(_,W)):-ϑ_screensize=:N,S+W<N-10,print(X),!.
ω(S,X,W):-ϑ_screensize=:N,0.6*N<S,SS is S-N//2,ω(SS,X,W),!.

ω(S,C:A#B,w(w(_,CC):w(AA#BB,_),_)):-
    ϑ_shift=:N,S1 is S+CC+1, S2 is S+N,
    print(C),write(':'),ω'(S1,A,AA),write('#'),nl,tab(S2), ω(S2,B,BB),!.
ω(S,C:A=>B,w(w(_,CC):w(AA=>BB,_),_)):-
    ϑ_shift=:N,S1 is S+CC+1, S2 is S+N,
    print(C),write(':'),ω'(S1,A,AA),write('=>'),nl,tab(S2),ω(S2,B,BB),!.
ω(S,C:A\B,w(w(_,CC):w(AA\BB,_),_)):-
    ϑ_shift=:N,S1 is S+CC+1, S2 is S+N,
```

```
      print(C),write(':'),ω′(S1,A,AA),write('\\'),nl,tab(S2), ω(S2,B,BB),!.
ω(S,C:A,w(w(_,CC):AA,_)):-S1 is S+CC+1,
      print(C),write(':'),ω′(S1,A,AA),!.
ω(S,A#B,w(AA#BB,_)):-
      ω′(S,A,AA),write('#'),nl,tab(S), ω(S,B,BB),!.
ω(S,A=>B,w(AA=>BB,_)):-
      ω′(S,A,AA),write('=>'),S1 is S+2,nl,tab(S1), ω(S1,B,BB),!.
ω(S,X,W):-ω′(S,X,W).



ω′(_,X,_) :- portray(X),!.
ω′(S,C:A#B,W):-par(ω(S,C:A#B,W)).
ω′(S,C:A=>B,W):-par(ω(S,C:A=>B,W)).
ω′(S,C:A,W):-par(ω(S,C:A,W)).

ω′(S,X,w(_,W)):-ϑ_screensize=:N,S+W<N-10,par(write(X)),!.
ω′(S,X,W):-ϑ_screensize=:N,0.6*N<S,SS is S-N//2,ω′(SS,X,W),!.

ω′(S,X,w(XX,_)):-
      append(Y,[T],X),append(YY,[TT],XX), S1 is S+1,
      write('['),weight(Y,w(_,Yw)),
      ( ϑ_screensize=:N,S+Yw+2>N,ω″(S1,Y,YY); ω″(Y) ),
      write(','),nl,tab(S1), ω′(S1,T,TT),write(']'),!.
ω′(S,lambda(X,F),w(lambda(_,FF),_)):-
      write('lambda('),write(X),write(','),
      S1 is S+2,nl,tab(S1),ω′(S1,F,FF),write(')'),!.
ω′(S,A#B,w(AA#BB,_)):-
      ω′(S,A,AA),write('#'),nl,tab(S), ω′(S,B,BB),!.
ω′(S,A=>B,w(AA=>BB,_)):-
      ω′(S,A,AA),write('=>'),S1 is S+2,nl,tab(S1), ω′(S1,B,BB),!.
ω′(S,A=B in T,w(w(AA=BB,_) in TT,_)):-
      S1 is S+1,S2 is S+3,S3 is S+4,
      write('('),ω′(S1,A,AA),nl,tab(S),
      write(' = '),ω′(S2,B,BB),nl,tab(S),
      write(' in '),ω′(S3,T,TT),write(')'),!.
ω′(S,A in T,w(AA in TT,_)):- S1 is S+1,S2 is S+4,
      write('('),ω′(S1,A,AA),nl,tab(S),
      write(' in '),ω′(S2,T,TT),write(')'),!.
ω′(S,{X},w({XX},_)):-S1 is S+1,
      write('{'),ω(S1,X,XX),write('}'),!.

ω′(_,u(I),_):-write(u(I)),!.
ω′(_,X,_):-atom(X),write(X),!.
```

```
ω′(_,X,_):-integer(X),write(X),!.
ω′(S,X,w(XX,_)):-
    standardform(X),X=..[F|A], write(F),write('('),nl,tab(S),
    XX=..[F|AA],ω″(S,A,AA),write(')'),!.
ω′(S,X,w(XX,_)):-
    X=..[F,A,B], XX=..[F,AA,BB],
    ϑ_shift=:N,weight(F,w(_,Fw)),S1 is S+1,S2 is S+N+Fw+1,
    write('('),ω′(S1,A,AA),nl,tab(S),
    tab(N),write(F),write(' '),ω′(S2,B,BB),write(')'),!.
ω′(_,X,_):-par(write(X)).


ω″([X]):-write(X).
ω″([X|Y]):-write(X),write(','),ω″(Y).
ω″(S,[X],[XX]):-ω′(S,X,XX).
ω″(S,[X|Y],[XX|YY]):-ω′(S,X,XX),write(','),nl,tab(S),ω″(S,Y,YY).


standardform(X):-var(X),!.
standardform([]):-!.
standardform([_|_]):-!.
standardform(X):-atom(X).
standardform(X):-integer(X),0=<X.
standardform(u(_)).
standardform(X):- functor(X,F,_),
  \+ member(F, [==>,:,=>,#,\,///,{},in,=,<,+,-,*,/,mod,&,::,of,list,
        then,or,repeat,complete,try]).


par(write(X)):-standardform(X),print(X),!.
par(ω′(S,X,W)):-standardform(X),ω′(S,X,W),!.
par(ω(S,X,W)):-standardform(X),ω(S,X,W),!.

par(write(X)):-write('('),print(X),write(')'),!.
par(X):-X=..[F,S,T,W],SS is S+1,XX=..[F,SS,T,W],write('('),call(XX),write(')').


weight(X,w(X,2)):-var(X),!,vars=:I,II is I+1,vars:=II,X=var(II),var(II):=0.
weight(X,w(X,Xw)):-atom(X),atom_codes(X,Y),length(Y,Xw),!.
weight(X,w(X,1)):-integer(X),X<10,!.
weight(X,w(X,Xw)):-integer(X),Y is (X//10),weight(Y,w(Y,Yw)),Xw is Yw+1,!.
```

```
weight([],w([],2)):-!.
weight([X],w([w(XX,Xw)],W)):-weight(X,w(XX,Xw)),W is Xw+2,!.
weight([X|Y],w([w(XX,Xw)|YY],W)):-weight(X,w(XX,Xw)),weight(Y,w(YY,Yw)),W is Xw+Yw+1,!.
weight(X,w(XX,Xw)):-
    X=..[F|A],weight(F,w(F,Fw)),length(A,LA),
    weight(A,w(AA,Aw)),XX=..[F|AA],Xw is Fw+Aw-LA+1.


prepare(var(I),'_'):-var(I)=:0,!.
prepare(var(I),V):-decimal(I,L),underscore(U),atom_codes(V,[U|L]),!.
prepare(atom(A),atom(AA)):-atom_codes(A,L),quote(L,LL),atom_codes(AA,LL).
prepare([],[]):-!.
prepare([H|T],[HH|TT]):-!,prepare(H,HH),prepare(T,TT).
prepare(X,XX):-X=..[F|A],prepare(A,AA),XX=..[F|AA].


quote([],[X,X]):-apostroph(X).
quote([X|T],[X,X|TT]):-apostroph(X),quote(T,TT).
quote([Y|T],[X,Y|TT]):- \+ apostroph(Y),quote(T,[X|TT]).


fringe(_,X,X):-var(X),!.
fringe(_,X,X):-atom(X),!.
fringe(_,X,X):-integer(X),!.
fringe(_,u(I),u(I)):-!.
fringe(_,[],[]):-!.
fringe(S,[T],[TT]):-fringe(S,T,TT),!.
fringe(S,[X|Y],[XX|YY]):-fringe(S,X,XX),fringe(S,Y,YY),!.
fringe(S,X,XX):-
  0<S,X=..[F|A],SS is S-1,fringe(SS,A,AA),XX=..[F|AA],!.
fringe(_,_,'___').
```

3.5 Substitutions

A substitution is described by the predicate $s(t,[y,...],[x,...],t')$ which states the fact, that the term t' may be obtained from t by substituting all occurrences of the variables $x,...$ by the corresponding subterms $y,...$ renaming bound variables in conflicting situations. Substitutions are performed simultaneously.

```
s(V1,V2,V3,V4) :- var(V1),var(V2),var(V3),var(V4),!,fail.
s(Tm, New, Old, SubdTm ) :-
```

```
    (bagof(Free, Sub^(nonvar(New), member(Sub,New), freevarsinterm(Sub,Free)), Frees), !
    ; Frees = []),
    freevarsinterm(Tm,ExtraFrees),
    union([ExtraFrees|Frees], SubFrees),
    pairing_of(Old, New, Insts),
    σ(Tm, Insts, SubFrees, [], SubdTm).


pairing_of([H1|T1], [H2|T2], [(H1 - H2)|T3]) :- pairing_of(T1, T2, T3).
pairing_of([],[],[]).
```

If term is one which is to be substituted, then replace it, as long as it is not a variable which is bound:

```
σ(Var,_,_,_,Var) :- var(Var),!.
σ(Term, _ , _ , Bound, Term) :- member(Term,Bound),!.
σ(Term, Insts, _ , Bound, Instd) :-
    member((Term - Instd), Insts), !, \+ member(Term, Bound).
σ(σ(Term,New,Old),Insts,Subfrees,Bound,SS) :-
    s(Term,New,Old,TT),σ(TT,Insts,Subfrees,Bound,SS).
σ(atom(Name),_,_,_,atom(Name)).
σ(term_of(Name),_,_,_,term_of(Name)).
σ({Name},_,_,_,{Name}) :- atom(Name).
σ({Var:Type\Pred}, Insts, SubFrees, Bound, {Avar:Stype\Spred}) :-
    member(Var, SubFrees), σ(Type, Insts, SubFrees, Bound, Stype),
    modify(Var, Avar), \+ member(Avar, SubFrees), !,
    σ(Pred, [(Var-Avar)|Insts], [Avar|SubFrees], Bound, Spred).
σ({Var:Type\Pred}, Insts, SubFrees, Bound, {Var:Stype\Spred}) :-
    σ(Type, Insts, SubFrees, Bound, Stype),
    σ(Pred, Insts, SubFrees, [Var|Bound], Spred).
σ((Var:Type#Pred), Insts, SubFrees, Bound, (Avar:Stype#Spred)) :-
    member(Var, SubFrees), σ(Type, Insts, SubFrees, Bound, Stype),
    modify(Var, Avar), \+ member(Avar, SubFrees),
    !, σ(Pred, [(Var-Avar)|Insts], [Avar|SubFrees], Bound, Spred).
σ((Var:Type#Pred), Insts, SubFrees, Bound, (Var:Stype#Spred)) :-
    σ(Type, Insts, SubFrees, Bound, Stype),
    σ(Pred, Insts, SubFrees, [Var|Bound], Spred).
σ((Var:Type=>Pred), Insts, SubFrees, Bound, (Avar:Stype=>Spred)) :-
    member(Var, SubFrees),!, σ(Type, Insts, SubFrees, Bound, Stype),
    modify(Var, Avar), \+ member(Avar, SubFrees),!,
    σ(Pred, [(Var-Avar)|Insts], [Avar|SubFrees], Bound, Spred).
σ((Var:Type=>Pred), Insts, SubFrees, Bound, (Var:Stype=>Spred)) :-
```

```
    !,σ(Type, Insts, SubFrees, Bound, Stype),
    σ(Pred, Insts, SubFrees, [Var|Bound], Spred).
σ(lambda(Var,Pred), Insts, SubFrees, Bound, lambda(Avar,Spred)) :-
    member(Var, SubFrees), modify(Var, Avar), \+ member(Avar, SubFrees),
    !, σ(Pred, [(Var-Avar)|Insts], [Avar|SubFrees], Bound, Spred).
σ(lambda(Var,Pred), Insts, SubFrees, Bound, lambda(Var,Spred)) :-
    σ(Pred, Insts, SubFrees, [Var|Bound], Spred).
σ(rec(Var,Pred), Insts, SubFrees, Bound, rec(Avar,Spred)) :-
    member(Var, SubFrees), modify(Var, Avar), \+ member(Avar, SubFrees),
    !, σ(Pred, [(Var-Avar)|Insts], [Avar|SubFrees], Bound, Spred).
σ(rec(Var,Pred), Insts, SubFrees, Bound, rec(Var,Spred)) :-
    σ(Pred, Insts, SubFrees, [Var|Bound], Spred).
σ([Bind], Insts, SubFrees, Bound, [SBind]) :-
    !, σ(Bind, Insts, SubFrees, Bound, SBind).
σ([~|Binding], Insts, SubFrees, Bound, [~|SBinding]) :-
    !, σ(Binding, Insts, SubFrees, Bound, SBinding).
σ([Var|Bind], Insts, SubFrees, Bound, [Avar|SBind]) :-
    member(Var, SubFrees), modify(Var, Avar), \+ member(Avar, SubFrees),
    !, σ(Bind, [(Var-Avar)|Insts], [Avar|SubFrees], Bound, SBind).
σ([Var|Bind], Insts, SubFrees, Bound, [Var|SBind]) :-
    !, σ(Bind, Insts, SubFrees, [Var|Bound], SBind).
```

If no binding required, and not something to be substituted simply substitute its arguments (if any):

```
σ(Term, _, _, _, Term) :- atomic(Term), !.
```

For compound terms, iterate over consituent parts of either first or last argument, depending on which one is instantiated, but exclude iteration over compound terms that have been treated as special cases above.

```
σ(Term, Insts, SubFrees, Bound, STerm) :-
    \+ var(Term),
    Term =.. [Funct|Args],
    \+ member(Funct, [su,atom,term_of,{},:,lambda,rec,~]),
    !,
    σ'(Args, Insts, SubFrees, Bound, SArgs),
    (var(STerm) ->
    (\+ \+ unify(STerm,SArgs),
      STerm =.. [Funct|SArgs]);
    (STerm =.. STermStruct,
      unify(STermStruct, [Funct|SArgs]))).
σ(Term, Insts, SubFrees, Bound, STerm) :-
    \+ var(STerm),
```

```
    STerm =.. [SFunct|SArgs],
    \+ member(SFunct, [su,atom,term_of,{},:,lambda,rec,~]),
    σ'(Args, Insts, SubFrees, Bound, SArgs),
    (var(Term) ->
    (\+ \+ unify(Term,Args),
      Term =.. [SFunct|Args]);
    (Term =.. TermStruct,
      unify(TermStruct, [SFunct|Args]))).


σ'([],_,_,_,[]).
σ'([H|T], Insts, SubFrees, Bound, [HH|TT]) :-
    σ(H,Insts, SubFrees, Bound, HH),
    σ'(T, Insts, SubFrees, Bound, TT).


slist([],_,_,[]).
slist([H|T],Y,X,[HH|TT]) :- s(H,Y,X,HH), slist(T,Y,X,TT).


shyp(H0,H==>G,Y,X,HH==>GG):-shyp(H0,H,Y,X,HH),s(G,Y,X,GG),!.
shyp(_,[],_,_,[]).
shyp(H,[_:D|T],Y,X,TT):- s(D,Y,X,DD),α(D,DD),!,shyp(H,T,Y,X,TT).
shyp(H,[_:D|T],Y,X,[VV:DD|TT]):-
  s(D,Y,X,DD),free([VV],H,HH),!,shyp(HH,T,Y,X,TT).
shyp(H,[_:D|T],Y,X,[VV:DD|TT]):-
  s(D,Y,X,DD),free([VV],H,HH),!,shyp(HH,T,Y,X,TT).
shyp(H,[_|T],Y,X,TT):-shyp(H,T,Y,X,TT).
```

3.6 Free variables

A variable is free in a term if it is a subterm of the term, and is not bound by the product, subset, or function types, or in a lambda or decision or induction term. $freevarinterm(Term, Var)$ succeeds if Var is free in $Term$, $freevarsinterm(Term, Vars)$ succeeds if $Vars$ is the set of free variables in $Term$. $appears(X, Y)$ is a predicate which tests whether X is a subterm of Y or not. It may be used for generating subterms of a certain pattern. $modify(v, v')$ modifies the variable name v by appending an underscore.

```
freevarsinterm(Tm, Vars) :- setof(Var, freevarinterm(Tm,Var), Vars), !.
freevarsinterm(_, []).
```

A Prolog Variable:

```
freevarinterm(X, _) :- var(X), !, fail.
```

Atomic terms non-atomic in prolog

```
freevarinterm(term_of(_),_) :- !,fail.
freevarinterm(atom(_),_) :- !,fail.
freevarinterm({N},_) :- atom(N),!,fail.
```

The binding term from decision or induction terms:

```
freevarinterm([H|T], Var) :-
    !, append(BoundVars, [Term], [H|T]), freevarinterm(Term, Var),
    \+ member(Var, BoundVars).
```

The binding types:

```
freevarinterm((_:T1#_), Var) :- freevarinterm(T1,Var).
freevarinterm((V:_#T2), Var) :- !, freevarinterm(T2,Var), \+ Var = V.
freevarinterm((_:T1=>_), Var) :- freevarinterm(T1,Var).
freevarinterm((V:_=>T2), Var) :- !, freevarinterm(T2,Var), \+ Var = V.
freevarinterm(({_:T1\_}), Var) :- freevarinterm(T1,Var).
freevarinterm(({V:_\T2}), Var) :- !, freevarinterm(T2,Var), \+ Var = V.
freevarinterm(rec(V,T), Var) :- !, freevarinterm(T, Var), \+ V = Var.
```

lambda term:

```
freevarinterm(lambda(V,T), Var) :- !, freevarinterm(T, Var), \+ V = Var.
freevarinterm(Var,Var) :- τ_var(Var).
```

Non-binding connectives etc:

```
freevarinterm(Tm, Var) :- Tm =.. [_|Args], member(Arg,Args), freevarinterm(Arg, Var).
```

```
appears(_,X):-var(X),!,fail.
appears(Y,X):-α(Y,X),!.
appears(_,[]):-!,fail.
appears(Y,[H|T]):-!,( appears(Y,H); appears(Y,T) ).
appears(Y,X):-X=..[F|A],(F==Y ; appears(Y,A)).
```

```
modify(X,Y):-atom_codes(X,XX),underscore(Z),append(XX,[Z],YY),atom_codes(Y,YY).
modify(X,Z):-modify(X,Y),modify(Y,Z).
```

3.7 Convertible Terms

Two types are called convertible if they are identical except possibly for renaming of bound variables. The scope of implicitly bound variables (as for example inside induction terms) is always given as a Prolog list, the last element of which is the term under consideration. This identifies `X:A=>B` and `A=>B` where X does not occur free in B.

```
α(X,Y):-(var(X);var(Y)),X=Y,!.
α(X,Y):- ground(X), ground(Y), X=Y, !.
α(X,Y):-(atom(X);atom(Y)),X=Y,!.
α(X,Y):-(integer(X);integer(Y)),X=Y,!.
α([X,T],[XX,TT]):- free([V],[X:dummy,T:dummy,XX:dummy,TT:dummy]),
   s(T,[V],[X],T0),s(TT,[V],[XX],TT0),!,α(T0,TT0).
α([X,Y,T],[XX,YY,TT]):-
free([U,V],[X:dummy,Y:dummy,T:dummy,XX:dummy,YY:dummy,TT:dummy]),
   s(T,[U,V],[X,Y],T0),s(TT,[U,V],[XX,YY],TT0),!,α(T0,TT0).
α([X,Y,Z,T],[XX,YY,ZZ,TT]):-
free([U,V,W],
[X:dummy,Y:dummy,Z:dummy,T:dummy,XX:dummy,YY:dummy,ZZ:dummy,TT:dummy]),
s(T,[U,V,W],[X,Y,Z],T0),s(TT,[U,V,W],[XX,YY,ZZ],TT0),!,α(T0,TT0).

α(rec(X,A),rec(XX,AA)):-!,α([X,A],[XX,AA]).
α(lambda(X,A),lambda(XX,AA)):-!,α([X,A],[XX,AA]).

α(X:A#B,XX:AA#BB):-!,α(A,AA),α([X,B],[XX,BB]).
α(X:A=>B,XX:AA=>BB):-!,α(A,AA),α([X,B],[XX,BB]).
α(X:A=>B,AA=>BB):-!,\+ freevarinterm(B,X),α(A=>B,AA=>BB).
α(A=>B,X:AA=>BB):-!,\+ freevarinterm(BB,X),α(A=>B,AA=>BB).
α({X:A\B},{XX:AA\BB}):-!,α(A,AA),α([X,B],[XX,BB]).
α(A=B in T,AA=BB in TT):-!,α(T,TT), (α(A,AA),α(B,BB); α(A,BB),α(B,AA)),!.
α(A in T, AA=BB in TT):-α(T,TT), α(A,AA),α(AA,BB).
α(AA=BB in TT, A in T):-α(T,TT), α(A,AA),α(AA,BB).

α(X,XX):-X=..[F|A],XX=..[F|AA],α'(A,AA).


α'([],[]).
α'([A|L],[AA|LL]):-α(A,AA),α'(L,LL).
```

3.8 Support for Peano Natural Numbers

Simple support functions for peano natural numbers..

```
pnat( s(X) ) :- pnat(X).
pnat( 0 ).
pnat( X ) :- nonvar(X),!,fail.
pdecide(s(X),s(Y),R) :-
     pdecide(X,Y,R).
pdecide(0,0,equal).
pdecide(0,s(_),less).
pdecide(s(_),0,more).
pless( X,Y ) :- pdecide(X,Y,less).

s_subterm( S, S ) :- !.
s_subterm( s(T), S ) :-
     s_subterm( T , S).

s_strip( s(A)<*s(B), AA<*BB ) :-
     s_strip( A<*B, AA<*BB ),
     !.
s_strip( L,L ).
```

3.9 Legality of Recursive Types

Implements restrictions as in NuPRL

```
illegal_rec_type(rec(Z,T)) :-
          ( occurs_in_dom(Z,T);
            occurs_in_fn_app(Z,T);
            occurs_in_elim_form(Z,T) ),
            write(' Illegal recursive type ! '), nl.

occurs_in_dom(Z,T) :- appears_in( X=>_ , T ),
                           appears_in( Z, X ).

occurs_in_fn_app(Z,T) :- appears_in(_ of X,T),
                              appears_in(Z,X).

occurs_in_elim_form(Z,T) :-
                appears_in(p_ind(X,_,_),T),
                appears_in(Z,X).
occurs_in_elim_form(Z,T) :-
                appears_in(cv_ind(X,_),T),
                appears_in(Z,X).
occurs_in_elim_form(Z,T) :-
                appears_in(ind(X,_,_,_),T),
                appears_in(Z,X).
occurs_in_elim_form(Z,T) :-
                appears_in(list_ind(X,_,_),T),
                appears_in(Z,X).
occurs_in_elim_form(Z,T) :-
```

```
                appears_in(rec_ind(X,_),T),
                appears_in(Z,X).
occurs_in_elim_form(Z,T) :-
                appears_in(atom_eq(X,Y,_,_),T),
                appears_in_list(Z,[X,Y]).
occurs_in_elim_form(Z,T) :-
                appears_in(int_eq(X,Y,_,_),T),
                appears_in_list(Z,[X,Y]).
occurs_in_elim_form(Z,T) :-
                appears_in(less(X,Y,_,_),T),
                appears_in_list(Z,[X,Y]).
occurs_in_elim_form(Z,T) :-
                appears_in(atom_eq(X,Y,_,_),T),
                appears_in_list(Z,[X,Y]).
occurs_in_elim_form(Z,T) :-
                appears_in(pnat_eq(X,Y,_,_),T),
                appears_in_list(Z,[X,Y]).
occurs_in_elim_form(Z,T) :-
                appears_in(pless(X,Y,_,_),T),
                appears_in_list(Z,[X,Y]).
occurs_in_elim_form(Z,T) :-
                appears_in(decide(X,_,_),T),
                appears_in(Z,X).
occurs_in_elim_form(Z,T) :-
                appears_in(spread(X,_),T),
                appears_in(Z,X).

appears_in(Z,Z).
appears_in(Z,F) :- F=..[_|T], appears_in_list(Z,T).
appears_in_list(_,[]) :- !,fail.
appears_in_list(Z,[H|T]) :- (appears_in(Z,H) ; appears_in_list(Z,T)).
```

3.10 Type Checking

Although in the given logical framework the type checking problem in general is not decidable, there are a lot of cases appearing in every day proofs, where the type property may be derived automatically. $type(X, H, T)$ is a predicate which computes for a small class of terms X the type T under the assumptions H. $type0(...)$ is a more general predicate checking the consistency of type information.

```
type(V,H,T):-atom(V),decl(V:rec(Z,TT),H),s(TT,[rec(Z,TT)],[Z],T),!.
type(V,H,T):-atom(V),decl(V:T,H),!.
type(atom(_),_,atom).
type(I,_,int):-integer(I).
type(X of Y,H,T):-type(X,H,TT=>T),type0(Y,H,TT).
type(X::Y,H,T list):-type(X,H,T),type0(Y,H,T list).
type(X&Y,H,T#TT):-type(X,H,T),type(Y,H,TT).
```

```
type(X,_,int):-functor(X,F,2),member(F,[+,-,*,/,mod]).


type0(X,H,T):-type(X,H,T).
type0(nil,_,_ list).
type0(inl(X),H,T\_):-type(X,H,T).
type0(inr(X),H,_\T):-type(X,H,T).
```

3.11 Rewrite Rules

Term rewriting $X \mapsto Y$ is described by the $\varphi(X, Y)$ predicate.

```
φ(X,X) :- var(X), !.
φ(lambda(X,B) of (A),Z):-s(B,[A],[X],Z).
φ(spread(X&Y,[U,V,T]),Z):-s(T,[X,Y],[U,V],Z).
φ(decide(inl(A),[X,S],_),Z):-s(S,[A],[X],Z).
φ(decide(inr(A),_,[X,S]),Z):-s(S,[A],[X],Z).
φ(list_ind(nil,X,_),X).
φ(list_ind(A::B,S,[X,Y,U,T]),Z):- s(T,[A,B,list_ind(B,S,[X,Y,U,T])],[X,Y,U],Z).
φ(p_ind(0,X,_),X).
φ(p_ind(s(A),B,[X,Y,T]),Z):-s(T,[A,p_ind(A,B,[X,Y,T])],[X,Y],Z).
φ(cv_ind(R,[H,I,T]),Z):-
    genvar(V),\+ appears(V,[H,I,T,R]),
    s(T,[R,lambda(V,cv_ind(V,[H,I,T]))],[H,I],Z),!.
φ(wo_ind(R,[H,I,T]),Z):-
    genvar(V),\+ appears(V,[H,I,T,R]),
    s(T,[R,lambda(V,wo_ind(V,[H,I,T]))],[H,I],Z),!.
φ(rec_ind(R,[H,I,T]),Z):-
    genvar(V),\+
    appears(V,[H,I,T,R]),s(T,[lambda(V,rec_ind(V,[H,I,T])),R],[H,I],Z),!.
φ(pnat_eq(A,B,S,_),S):- pdecide(A,B,equal).
φ(pnat_eq(A,B,_,T),T):- pdecide(A,B,R), \+ R = equal.
φ(pless(A,B,S,_),S):- pdecide(A,B,less).
φ(pless(A,B,_,T),T):- pdecide(A,B,R), \+ R = less.
φ(ind(N,[X,Y,Z],B,S),T):-
    integer(N),N<0,NN is N+1,s(Z,[N,ind(NN,[X,Y,Z],B,S)],[X,Y],T).
φ(ind(0,_,B,_),B).
φ(ind(N,S,B,[U,V,W]),T):-
    integer(N),N>0,NN is N-1,s(W,[N,ind(NN,S,B,[U,V,W])],[U,V],T).
φ(atom_eq(atom(A),atom(B),S,_),S):-A=B.
φ(atom_eq(atom(A),atom(B),_,T),T):-A\==B.
φ(int_eq(A,B,S,_),S):-integer(A),integer(B),A=B.
φ(int_eq(A,B,_,T),T):-integer(A),integer(B),A=\=B.
φ(less(A,B,S,_),S):-integer(A),integer(B),A<B.
φ(less(A,B,_,T),T):-integer(A),integer(B),A>=B.
φ(-A, Z):-integer(A),Z is -A.
φ(0+B,B):-!.
φ(A+0,A):-!.
```

```
φ(A+B,Z):-integer(A),integer(B),Z is A+B.
φ(A-B,Z):-integer(A),integer(B),Z is A-B.
φ(A-0,A):-!.
φ(0*_,0):-!.
φ(_*0,0):-!.
φ(1*B,B):-!.
φ(A*1,A):-!.
φ(A*B,Z):-integer(A),integer(B),Z is A*B.
φ(A/1,A):-!.
φ(A/B,Z):-integer(A),integer(B),Z is A//B.
φ(A mod B,Z):-integer(A),integer(B),Z is A mod B.
φ(term_of(T),TT) :-
    ϑ_theorem(T) =: P,

    extract(P,TTraw),polish(TTraw,TT).
φ({F},TT) :-
    cdef(F) =: ({F}<==>TT).
φ(T,TT) :-
    T=..[F|A],
    \+ F = {},
    cdef(F) =: (T1<==>T2),
    T1=..[F|P],
    s(T2,A,P,TT).

φ(0,X,Z):-φ(X,Y),!,φ(0,Y,Z).
φ(0,X,X).
φ(1,X,Z):-φ(X,Z).
φ(N,X,Z):-integer(N),N>1,NN is N-1,φ(X,Y),φ(NN,Y,Z).
```

3.12 Evaluation

The evaluation predicate $eval(X, Y)$ allows the direct execution of extract terms or user supplied terms X in the Prolog environment, yielding the result value as second parameter Y. A typical example for using the evaluator is in defining executable Prolog predicates using the extract terms derived from a theorem proven inside Oyster. Evaluating a term requires first the evaluation of each subterm and then rewriting of the resulting term. It is useful to apply *eval* at least once after extracting the term and running it, just in the sense of a partial evaluator.

```
eval(X,X) :- var(X),!.
eval(lambda(X,B) of (A),R):-eval(B,[A],[X],R),!.
eval(X of Y, R) :- eval(X, XX), \+(X=XX), eval(XX of Y, R), !.

eval(spread(A,[U,V,T]),R):-
    eval(A,AA),
    (AA=(X&Y),eval(T,[X,Y],[U,V],R);
      R=spread(AA,[U,V,T])
```

```
    ),!.

eval(decide(A,[X,S],[Y,T]),C):-
  eval(A,AA),
  (AA=inl(B),eval(S,[B],[X],C);
    AA=inr(B),eval(T,[B],[Y],C);
    C=decide(AA,[X,S],[Y,T])
  ),!.

eval(list_ind(L,S,[X,Y,U,T]),C):-
  eval(L,LL),
  (LL=nil,eval(S,C);
    LL=A::B,eval(T,[A,B,list_ind(B,S,[X,Y,U,T])],[X,Y,U],C);
    evalcond(U,T,TT),C=list_ind(L,S,[X,Y,U,TT])
  ),!.

eval(p_ind(N,S,[X,Y,T]),C):-
  eval(N,NN),
  (NN=0,eval(S,C);
    NN=s(A),eval(T,[A,p_ind(A,S,[X,Y,T])],[X,Y],C);
    evalcond(Y,T,TT),C=p_ind(NN,S,[X,Y,TT])
  ),!.

eval(cv_ind(N,[X,Y,T]),C):-
  eval(N,NN),
  (pnat(NN),genvar(V), \+ appears(V,[N,X,Y,T]),
    eval(T,[NN,lambda(V,cv_ind(V,[X,Y,T]))],[X,Y],C);
    evalcond(Y,T,TT),C=cv_ind(NN,[X,Y,TT])
  ),!.

eval(wo_ind(N,[X,Y,T]),C):-
\+var(N),eval(N,Nx),
genvar(V),\+appears(V,[X,Y,T,Nx]),
s(T,[Nx,lambda(V,wo_ind(V,[X,Y,T]))],[X,Y],Cx),
eval(Cx,C),!.

eval(atom_eq(A,B,S,T),R):-
  eval(A,atom(X)),eval(B,atom(Y)), (X=Y->eval(S,R);eval(T,R)),!.
eval(int_eq(A,B,S,T),R):-
  eval(A,X),eval(B,Y),integer(X),integer(Y),(X=Y,eval(S,R);eval(T,R)),!.
eval(less(A,B,S,T),R):-
  eval(A,X),eval(B,Y),integer(X),integer(Y),(X<Y,eval(S,R);eval(T,R)),!.
eval(pnat_eq(A,B,S,T),RR):-
  eval(A,X),eval(B,Y),pdecide(X,Y,R),
  (R=equal,eval(S,RR);eval(T,RR)),!.
eval(pless(A,B,S,T),RR):-
  eval(A,X),eval(B,Y),pdecide(X,Y,R),
  (R=less,eval(S,RR);eval(T,RR)),!.

eval(-A, Z):-eval(A,X),φ(-X,Z),!.
```

```
eval(A+B,Z):-eval(A,X),eval(B,Y),φ(X+Y,Z),!.
eval(A-B,Z):-eval(A,X),eval(B,Y),φ(X-Y,Z),!.
eval(A*B,Z):-eval(A,X),eval(B,Y),φ(X*Y,Z),!.
eval(A/B,Z):-eval(A,X),eval(B,Y),φ(X/Y,Z),!.
eval(A mod B,Z):-eval(A,X),eval(B,Y),φ(X mod Y,Z),!.

eval(ind(N,[X,Y,Z],B,[U,V,W]),T):-
   eval(N,NN),
   (integer(NN),NN<0,N0 is NN+1,eval(Z,[NN,ind(N0,[X,Y,Z],B,S)],[X,Y],T);
      NN=0,eval(B,T);
      integer(NN),NN>0,N0 is NN-1,eval(W,[NN,ind(N0,S,B,[U,V,W])],[U,V],T);
      evalcond(Y,Z,ZZ),evalcond(V,W,WW),T=ind(NN,[X,Y,ZZ],B,[U,V,WW])
   ),!.

eval(rec_ind(R,[H,I,T]),Z):-
   eval(R,RR),
   genvar(V),\+ appears(V,[H,I,T,RR]),
   s(T,[lambda(V,rec_ind(V,[H,I,T])),RR],[H,I],ZZ),
   eval(ZZ,Z),!.

eval(X,Y):-
   X=..[F|A],hypothesis(D<==>E),D=..[F|B],eval(E,A,B,Y).

eval({F},Z) :-
      atom(F),
      !,
      cdef(F) =: ({F}<==>TT),
      eval(TT,ZZ),
      eval(ZZ,Z).

eval(T,Z) :-
      T=..[F|A],
      cdef(F) =: (T1<==>T2),
      !,
      T1=..[F|P],
      evallist(A,B),
      s(T2,B,P,ZZ),
      eval(ZZ,Z).

eval(term_of(T),Z):-
      ϑ_theorem(T) =: P,
      extract(P,Eraw),
      \+var(Eraw),
      polish(Eraw,E),
      eval(E,Z).

eval(X,Y):-
   X=..[F|A],
   \+ member(F,[atom,lambda,ind,list_ind,rec_ind,wo_ind,pnat_eq,atom_eq,int_eq,less,pless]),
   evallist(A,B),Y=..[F|B],!.
```

```
eval(X,X).

eval(T,L1,L2,TT) :- s(T,L1,L2,T1), eval(T1,TT).


evallist([],[]).
evallist([X|XX],[Y|YY]):-eval(X,Y),evallist(XX,YY).


evalcond(V,T,T):-appears(V,T),!.
evalcond(_,T,TT):-eval(T,TT).
```

3.13 Computation of tagged terms $[[t]]$

The compute rule has been generalised to get a simple user interface to all different kinds of possible and sound rewritings. The use of anonymous variables allows you to use partial specifications of the term structure. $compute(T, X, Y, H, S)$ describes the process of computing X guided by some *tagged term* T under the assumption of the *hypotheses* H, which results in Y and possibly a set of subgoals which have to be proven to satisfy this computation process.

```
compute_old(X,T,T):-var(X),!.

compute_old([[N]],T,TT):-integer(N),φ(N,T,TT).
compute_old([[simplify]],T,TT):-eval(T,TT), \+ T=TT,!.
compute_old([[unfold]],T,TT) :-
    φ(1,T,TT).
compute_old([[expand]],T,TT) :-
    φ(1,T,TT).
compute_old([[fold(F)]],B,{F}) :-
cdef(F)=:({F}<==>B).
compute_old([[fold(F)]],T,TT):-
    cdef(F)=:(T1<==>T2),
    T1=..[F|P],s(T2,X,P,T0),T0=T,s(T1,X,P,TT).

compute_old(T,X,XX):-
    T=..[F|A],X=..[F|B],compute_list_old(A,B,BB),XX=..[F|BB].


compute_list_old([],[],[]).
compute_list_old([T|TT],[X|XX],[Y|YY]):-
    compute_old(T,X,Y),compute_list_old(TT,XX,YY).
```

```
compute_using([[N,T]],TT):-
    integer(N),
    !,
    compute_using(T,CT),
    φ(N,CT,TT).

compute_using([[unfold,T]],TT):-
    !,
    compute_using(T,CT),
    φ(1,CT,TT).
compute_using([[fold(F),T]],TT):-
    cdef(F)=:(T1<==>T2),T1=..[F|P],s(T2,X,P,T0),T0=T,s(T1,X,P,TT).
compute_using([[unroll,T]],TT):-
    !,
    compute_using(T,CT),
    φ(1,CT,TT).

compute_using([[expand,T]],TT):-
    !,
    φ(1,T,TT).

compute_using([H|T], TT) :-
    !,
    append( Vars, [Body], [H|T] ),
    compute_using( Body, CBody ),
    append( Vars, [CBody], TT ).
compute_using([],_):-!,fail.

compute_using(T,XX):-
    imm_subterms(T,A,F),
    compute_list(A,BB),
    \+ A = BB,
    imm_subterms(XX,BB,F),
    !.
compute_using(T,T).



compute_list_save( L,R ) :-
    compute_list( L,R),
    \+ L = R,
    !.
compute_list_save( L,L ).

compute_list([],[]).
compute_list([T|TT],[Y|YY]):-
    compute_using(T,Y),
    compute_list(TT,YY).

normalise( Ilgl, T, NT ) :-
```

```
    correct_tag(Ilgl,T,TT),
    \+ T=TT,
    compute_using(TT,CT),
    !,
    normalise(Ilgl,CT,NT).
normalise(_,T,T).
```

3.14 Generate / Check tagging for compute with using() clause

$correct\_tag(Ignore, Term, TaggedTerm)$ holds if TaggedTerm is a correct tagging of Term. That is, if TaggedTerm is Term tagged in places where computation rules apply. Where TaggedTerm is not bound, it becomes bound to Term with all legal tags that do not matching those in Ignore.

```
correct_tag( _, Tm, _ ) :-
    var(Tm),
    !.
correct_tag( Ilgl, Tm, TagTm ) :-
    verify_tagging( Ilgl, TagTm, Tm, Tagged ),
    imm_subterms( Tm, SubTms, TmId ),
    imm_subterms( Tagged, TagSubTms, TmId ),
    !,
    correct_tag_list( Ilgl, SubTms, TagSubTms ).

correct_tag_list( Ilgl, [SubTm|RestSubTms], [TagSubTm|RestTagSubTms] ) :-
    SubTm = [_|_],
    !,
    append( Vars, [BoundSubTm], SubTm ),
    append( Vars, [BoundTagSubTm], TagSubTm ),
    correct_tag( Ilgl, BoundSubTm, BoundTagSubTm ),
    correct_tag_list( Ilgl, RestSubTms, RestTagSubTms ).
correct_tag_list( Ilgl, [SubTm|RestSubTms], [TagSubTm|RestTagSubTms] ) :-
    correct_tag( Ilgl, SubTm, TagSubTm ),
    correct_tag_list( Ilgl, RestSubTms, RestTagSubTms ).
correct_tag_list( _, [],[] ).
```

, Check/generate a legal tag for computing a non-canonical term.

```
verify_tagging( Ilgl, TagTm, Tm, Tagged ) :-
    nonvar( TagTm ),
    ( ( TagTm = [[Tag,Tagged]],
        !,
        tag_for( Ilgl, Tm, Tag )
```

```
        );
        TagTm = Tagged
    ),
    !.

verify_tagging( Ilgl, TagTm, Tm, Tagged ) :-
    tag_for( Ilgl, Tm, Tag ),
    TagTm = [[Tag,Tagged]].

verify_tagging( _, Tagged, _, Tagged ).
```

Non-canonical and defined terms and their appropriate tags

```
tag_for( Ilgl, Tm, Tag ) :-
    tag_for( Tm, GTag ),
    !,
    \+ member( [GTag,Tm], Ilgl ),
    ( (Tag = GTag);
      (integer(Tag),integer(GTag))
    ).

tag_for( (lambda(_,_) of _ ), 1 ).
tag_for( decide(inl(_),_,_), 1 ).
tag_for( decide(inr(_),_,_), 1 ).
tag_for( spread(_&_,_), 1 ).
tag_for( p_ind(0,_,_), 1 ).
tag_for( p_ind(s(_),_,_), 1 ).
tag_for( rec_ind(A,_), 1 ) :-
    \+ atom(A),
    functor(A,F,_),
    member(F,[inl,inr,s,0,'&']),
    !.
tag_for( cv_ind(A,_), 1 ) :-
    pnat(A),!.
tag_for( wo_ind(A,_), 1 ) :-
    \+ atom(A),
    functor(A,F,_),
    member(F,[inl,inr,s,0,'&']),
    !.
tag_for( cv_ind(_,_), unroll ).
tag_for( wo_ind(_,_), unroll ).
tag_for( rec_ind(_,_), unroll ).
tag_for( rec(_), 1 ).
tag_for( atom_eq(atom(_),atom(_),_,_), 1 ).
tag_for( (_+0), 1 ).
tag_for( (0+_), 1 ).
tag_for( (0*_), 1 ).
tag_for( (_*0), 1 ).
```

```
tag_for( (1*_), 1 ).
tag_for( (_*1), 1 ).
tag_for( (_-0), 1 ).
tag_for( (_/1), 1 ).
tag_for( ind(A,_,_,_), 1 ) :- integer(A).
tag_for( (A + B), 1 ) :- integer(A),integer(B).
tag_for( (A / B), 1 ) :- integer(A),integer(B).
tag_for( (A * B), 1 ) :- integer(A),integer(B).
tag_for( (A - B), 1 ) :- integer(A),integer(B).
tag_for( (A mod B), 1 ) :- integer(A),integer(B).
tag_for( less(A,B,_,_), 1 ) :- integer(A),integer(B).
tag_for( int_eq(A,B,_,_), 1 ) :- integer(A),integer(B).
tag_for( pless(A,B,_,_), 1 ) :- pdecide(A,B,_).
tag_for( pnat_eq(A,B,_,_), 1 ) :- pdecide(A,B,_).

tag_for( term_of(_), expand ).
tag_for( {N}, unfold ) :-
    atom(N),
    !,
    (cdef(N) =: _).
tag_for( Tm, unfold ) :-
    functor(Tm,N,_),
    cdef(N) =: _.
```

Break up/rebuild a term into/from its subterms and a token identifying its kind.

```
imm_subterms( atom(A), [], atom(A) ) :-
    !.

imm_subterms( term_of(Name), [], term_of(Name) ) :-
    !.

imm_subterms( {Name}, [], {Name} ) :-
    atom(Name),
    !.

imm_subterms( (V:T1#T2), [T1,[V,T2]], 'b#' ) :- % *** Deal with binding types
    !.
imm_subterms( (V:T1=>T2), [T1,[V,T2]], 'b=>' ) :- !.

imm_subterms( ({V:T1|T2}), [T1,[V,T2]], 'b{}' ) :- !.

imm_subterms( lambda(V,T), [[V,T]], 'lambda' ) :- !. % *** lambda term

imm_subterms( rec(V,T), [[V,T]], 'rec' ) :- !. % *** lambda term

imm_subterms( {A|B}, [A,B], 'n{}' ).

imm_subterms( A in B, [A,B], 'in1' ).

imm_subterms( A=B in C, [A,B,C], 'in2' ).
```

```
imm_subterms(Dterm, [], Dterm) :-
    τ_var(Dterm),
    !.

imm_subterms( Tm, Args, (Funct/Arity) ) :-
    nonvar(Tm),
    Tm =.. [Funct|Args],
    length(Args,Arity).

imm_subterms( Tm, Args, (Funct/Arity) ) :-
    var(Tm),
    length(Args,Arity),
    Tm =.. [Funct|Args].
```

3.15 Manipulations of the hypothesis list

$thin\_hyps(ToThin, Hyps, ThinnedHyps)$ gives $Hyps$ after removing the hypothesis named in $ToThin$ as $ThinnedHyps$.

$extend\_thin(ToThin, Hyps, ExtToThin$ gives in $ExtToThin$ all the hypotheses (by number) that need to be thinned from $Hyps$ as a result of thinning those named in $ToThin$.

$replace\_hyps(ToRepl, Hyps, ReplHyps)$ Gives in $ReplHyps$ the result of replacing in $Hyps$ those hypotheses with a hypothesis of the same name in $ToRepl$.

```
thin_hyps( ToThin, [(Name:_)|RestHyps], Thinned ) :-
    member(Name,ToThin),
    !,
    thin_hyps( ToThin, RestHyps, Thinned ).
thin_hyps( ToThin, [Hyp|RestHyps], [Hyp|Thinned] ) :-
    thin_hyps( ToThin, RestHyps, Thinned ).
thin_hyps( _, [], [] ).

extend_thin( ToThin, [(Name:Body)|RestHyps], NotThin, ExtToThin ) :-
    member(Name,NotThin),
    !,
    freevarsinterm( Body, Frees ),
    union(NotThin,Frees,NextNotThin),
    extend_thin( ToThin, RestHyps, NextNotThin, ExtToThin ).
extend_thin( ToThin, [(Name:Body)|RestHyps], NotThin, ExtToThin ) :-
    freevarsinterm( Body, Frees ),
    member( F, Frees ),
    member( F, ToThin ),
    union([Name],ToThin,NextToThin ),
    !,
    extend_thin( NextToThin, RestHyps, NotThin, ExtToThin ).

extend_thin( ToThin, [_|RestHyps], NotThin, ExtToThin ) :-
    extend_thin( ToThin, RestHyps, NotThin, ExtToThin ).
```

```
extend_thin( ToThin, [], NotThin, ExtToThin ) :-
    subtract( ToThin, NotThin, ExtToThin ).

replace_hyps( ToReplace, [(Name:_)|RestHyps], [(Name:RBody)|RestRHyps] ) :-
    member( (Name:RBody), ToReplace ),
    !,
    replace_hyps( ToReplace, RestHyps, RestRHyps ).
replace_hyps( ToReplace, [Hyp|RestHyps], [Hyp|RestRHyps] ) :-
    replace_hyps( ToReplace, RestHyps, RestRHyps ).
replace_hyps( _,[], [] ).
```

3.16 Utilities

- List difference: $diff(X, Y, Z)$ computes the list Z of all elements of X, which are not in Y.

```
diff([],_,[]).
diff([X|T],Y,TT):-member(X,Y),!,diff(T,Y,TT).
diff([X|T],Y,[X|TT]):-diff(T,Y,TT).
```

Sound unification unify(X, Y) is true when the two terms X and Y unify *with* the occurs check.

```
unify(X,Y) :-
    ground(X),!,
    X = Y.
unify(X,Y) :-
    ground(Y),!,
    X = Y.
unify(X, Y) :-
    ( nonvar(X), nonvar(Y) ->
functor(X, F, N),
functor(Y, F, N),
unify(N, X, Y)
    ; nonvar(X) /* var(Y) */ ->
free_of_var(Y, X), % Y does not occur in X
Y = X
    ; nonvar(Y) /* var(X) */ ->
free_of_var(X, Y), % X does not occur in Y
X = Y
    ; /* var(X), var(Y) */
X = Y % unify(X, X) despite X
    ). % occurring in X!

unify(N, X, Y) :-
    ( N < 1 -> true
    ; arg(N, X, Xn),
arg(N, Y, Yn),
```

```
unify(Xn, Yn),
M is N-1,
unify(M, X, Y)
     ).
```

- generating new variables from the sequence $v0, v1, v2, ...$

```
genvar(X):-var(X),!,genint(Y),decimal(Y,Z),atom_codes(X,[118|Z]).
genvar(X):-τ_var(X).


genint(0).
genint(X):-genint(Y),X is Y+1.



decimal(Y,[Z]):-Y<10,digit(Y,Z),!.
decimal(Y,Z):-Y1 is (Y//10), Y2 is (Y mod 10),
     decimal(Y1,Z1),decimal(Y2,Z2),append(Z1,Z2,Z).
```

- generate or check universe level

```
level(I):-var(I),ϑ_universe=:I,!.
level(I):-integer(I),0<I.
```

- generate or check free variables from hypothesis list

```
free([],_).
free([X|L],H):-var(X),!,genvar(X),freevar(X,H),free(L,[X:_|H]),!.
free([X|L],H):-τ_var(X),freevar(X,H),free(L,[X:_|H]).

free(X,H,HH):-free(X,H),extend(X,H,HH).


extend([],H,H).
extend([X|T],H,[X:_|HH]):-extend(T,H,HH).



freevar(X,H):-decl(X:_,H),!,fail.
freevar(X,H):-member(D<==>_,H),functor(D,X,_),!,fail.
freevar(_,_).
```

- Checking whether an identifier is free in a problem, i.e. it is not declared neither in the current hypothesis list, nor in the nypothesis lists of any subproblem.

```
notused(_,S):-var(S),!.
notused(X,π(H==>_,_,_,S)):-free([X],H),notused(X,S).
notused(_,[]).
notused(X,[P ext _|S]):-notused(X,P),notused(X,S).
notused(X,[P|S]):-notused(X,P),notused(X,S).
```

- Declaration in hypothesis list

```
decl(X:D,H):-member(X:D,H).
```

- copying Prolog terms for generating fresh variables.

```
copy(T,TT):-recorda(copyterm,T,R),recorded(copyterm,TT,R),erase(R).
```

- Reading a term of given structure from an external file

```
readfile(F,X):- see(F),read(Y),X=Y,!,seen.
readfile(_,_):- seen,nl,write('wrong structured file.'),nl,!,fail.
```

- Tabbing; tab/1 is non-ISO; simulate it here

```
tab(N) :- on_exception( E , _ is N,
( print( "bad argument to tab/1" ),
throw(E) )
),
          ( N >= 0 ->
ttab(N)
             ; (print( 'negative argument to tab/1'),
throw(neg_arg)
                )
          ).


ttab(0).
ttab(N) :- N > 0, put_char( ' ' ), M is N-1, ttab(M).
```

- Code dependent predicates

```
small_letter(X):-97=<X,X=<122.

capital_letter(X):-65=<X,X=<90.

digit(X):-48=<X,X=<57.
digit(X,Y):-0=<X,X=<9,Y is X+48.

underscore(95).

apostroph(39).
```

- Toggle a global variable on execution and backtracking: the global variable Var is being set to $Val1$ on execution, and to $Val2$ on backtracking. Notice that $toggle/2$ does not give an extra backtrackpoint, since the second clause is forced to fail.

```
toggle(Var, [Val1,_]) :- Var:=Val1.
toggle(Var, [_,Val2]) :- Var:=Val2, fail.
```

References

[**1**] Brouwer,L.E.J.:
On the significance of the principle of excluded middle in mathematics, especially in function theory.
J.für Reine und Angewandte Mathematik **154** (1923),1-7

[**2**] Constable,R.L. et al:
Implementing Mathematics with the Nuprl Proof Development System,
Prentice Hall, Englewood Cliffs, 1986

[**3**] Heyting,A.:
Intuitionism: An Introduction,
North-Holland, Amsterdam, 1956

[**4**] Kolmogorov,A.N.:
Zur Deutung der intuitionistischen Logik.
Mathematische Zeitschrift, **35** (1932), 58-65

[**5**] Kowalski,R.:
Logic for Problem Solving,
North-Holland, New York, 1979

[**6**] Martin-Löf,P.:
Constructive mathematics and computer programming,
in: *6th Int.Congress for Logic, Methodlogy of Science, and Philosophy*,
North-Holland, Amsterdam, 1982, pp.153-175

[**7**] Martin-Löf,P.:
Intuitionistic Type Theory,
Bibliopolis, Napoli, 1984

[**8**] Russell,B.:
Mathematical logic based on a theory of types.
Am.J.of Math. **30** (1908), pp. 222-262

Appendix A

Examples

The aim of this appendix is to give a collection of examples which demonstrate different ways of using the Oyster system. These examples are not primarily intended as a tutorial, but more as illustration of some aspects of the system.

A.1 Factorial

In this section we give two examples for deriving the *faktorial* function `n!`. The first approach starts with a complete specification and in fact consists only of one main step supplying the full implementation. The second example shows how you can synthesise the factorial function from an underspecification (as a mapping of the type `int=>int`).

A.1.1 Verification

```
| ?- create_thm(fak,user).
|: []==>fak:(int=>int)#
         fak of 0=1 in int#
         n:int=>0<n=>
            fak of n=n*fak of (n-1) in int.

yes
| ?-
```

In the beginning we have to select the theorem:

```
| ?- select(fak),display.
fak : [] incomplete
==> fak:(int=>int)#
    (fak of 0=1 in int)#
    n:int=>0<n=>fak of n=n*fak of (n-1)in int
by _
```

```
yes
| ?-
```

The implementation step should be combined with a simplification. This makes the resulting subgoals more comprehensive:

```
| ?- intro(lambda(i,ind(i,[~,~,0],1,[i,f,i*f])))
       then try simplify.

yes
| ?- display.
fak : [] partial
==> fak:(int=>int)#
    (fak of 0=1 in int)#
    n:int=>0<n=>fak of n=n*fak of (n-1)in int
by intro(lambda(i,ind(i,[~,~,0],1,[i,f,i*f])))then try simplify

  [1] incomplete
  1. n:int
  2. v0:0<n
  ==> (ind(n,[~,~,0],1,[i,f,i*f])
     = (n*ind(n-1,[~,~,0],1,[i,f,i*f]))
     in int)

yes
| ?-
```

The pending subgoal can be resolved easily, if we use the appropriate *reduction* rule for integer induction terms.

```
| ?- down(1),reduce(left,up),status.
complete
yes
| ?-
```

At the end we have a look at the complete proof and its extract term:

```
|?- top,snapshot.
fak : [] complete
==> fak:(int=>int)#
    (fak of 0=1 in int)#
    n:int=>0<n=>fak of n=n*fak of (n-1)in int
by intro(lambda(i,ind(i,[~,~,0],1,[i,f,i*f])))then try simplify

  [1] complete
  1. n:int
  2. v0:0<n
  ==> (ind(n,[~,~,0],1,[i,f,i*f])
```

```
      = (n*ind(n-1,[~,~,0],1,[i,f,i*f]))
      in int)
  by reduce(left,up)

yes
| ?- extract.
lambda(i,ind(i,[~,~,0],1,[i,f,i*f]))
  & axiom
    & lambda(~,lambda(~,axiom))

yes
| ?-
```

To allow the direct execution of the Oyster program on the Prolog level we should generate a Prolog predicate, which evaluates the the function term:

```
| ?-extract(X& _),assert((fak(A,B):-eval(X of A,B))).

yes
| ?-
```

Running this Prolog predicate yields for example:

```
| ?- genint(I),fak(I,J),write(fak(I)=J),nl,I>9.
fak(0)=1
fak(1)=1
fak(2)=2
fak(3)=6
fak(4)=24
fak(5)=120
fak(6)=720
fak(7)=5040
fak(8)=40320
fak(9)=362880
fak(10)=3628800

yes
| ?-
```

A.1.2 Synthesis

```
| ?- create_thm(fak,user).
|: []==>int=>int.

yes
| ?- select(fak).
```

```
yes
| ?-
```

Program synthesis should be generally executed in the *pure* mode. The troubles comes from the filter rule sitting in the beginning of the rule base, which simply checks, whether the current goal is already part of the hypothesis list. This rule guarantees in normal use the automatic recognition of induction hypotheses and makes the standard autotactic *repeat intro* really smart, but for synthesis purposes you have to switch it off, because it would generate the most trivial program structure. If you want to synthesise an integer term, this rule would pick up the first variable declared to be of the type *int* and yield this variable as implementation. In this example *intro* would "prove" the goal completely automatically, yielding the extract term `lambda(v0,v0)`.

```
| ?- pure(intro).

yes
| ?- display.
fak : [] partial
==> int=>int
by pure(intro)

  [1] incomplete
  1. v0:int
  ==> int

  [2] incomplete
  ==> int in u(1)


yes
| ?- pure(intro),extract.
lambda(v0,_)

yes
| ?-
```

For pragmatic reasons it is useful to ignore all wellformedness goals during the synthesis process and tidy the proof tree up at the end. The interesting subgoal is the first one. Now we have to prove the existence of an integer starting from an integer *v0* already known. If you have no idea how to continue, simply call *apply(X)*, it will generate all rules applicable.

```
| ?- down(1).

yes
| ?- apply(X),write(X),nl,fail.
```

```
intro
intro(<integer>)
intro(- ~)
intro(~ + ~)
intro(~ - ~)
intro(~ * ~)
intro(~ / ~)
intro(~mod~)
elim(v0,new [v1,v2,v3])

no
| ?-
```

The corresponding extract terms would be: $v0$ - the hypothesis directly available, arbitrary integer numbers (as supplied), compound terms with the top level operators $-$ (unary) or $+$, $-$, $*$, $/$, and *mod* (binary) , or an integer induction term. We decide for the last one:

```
| ?- pure(elim(v0)).

yes
| ?- display.
fak : [1] partial
1. v0:int
==> int
by pure(elim(v0))

  [1] incomplete
  2. v1:int
  3. v2:v1<0
  4. v3:int
  ==> int

  [2] incomplete
  ==> int

  [3] incomplete
  2. v1:int
  3. v2:0<v1
  4. v3:int
  ==> int

yes
| ?- extract.
ind(v0,[v1,v3,_],_,[v1,v3,_])

yes
```

```
| ?-
```

Now we have to supply refinement terms for each of the three cases. The first one is trivial - for negative arguments we are free to choose any value, but we have to supply one value, say 0. Of course we could leave this case open and let it be filled in automatically by our tidying strategy at the end. For the zero case we choose 1. For the main case we have the rough idea of choosing a term of the form $n * f(n-1)$. The values for n and $f(n-1)$ are available as $v1$ and $v3$, so we give a direct implementation of the form $v1 * v3$:

```
| ?- down(1),pure(intro(0)),up.

yes
| ?- down(2),pure(intro(1)),up.

yes
| ?- down(3),pure(intro(explicit(v1*v3))),up.

yes
| ?-
```

At the end we have to "tidy up" and to prove all open wellformedness goals. In this simple case it would be enough to run around the proof tree, and try to apply the standard autotactic:

```
| ?- top,next,intro,fail.

no
| ?- status.
complete
yes
| ?-
```

Considering the extract term, we get essentially the same as for the fully specified function, except that there is no additional component describing the proof of the properties:

```
| ?- extract.
lambda(v0,ind(v0,[~,~,0],1,[v1,v3,v1*v3]))

yes
| ?-
```

If you are constructing fairly large programs, the normal strategy of solution would be a combination between both approaches. The *algorithmic idea* is an essential guideline for constructing good programs. That's why one could prefer to built a program by stepwise refinement (from time to time looking at the extract term) and then taking the extract term from that synthesising proof as input for the critical proof step of the verifying proof.

Index

$*$, 5
$+$, 5
$-$, 5
$/$, 5
$///$, 5
:, 5
::, 5
$<$, 5
$<=$, 5
$<==>$, 5
$=$, 5
$==>$, 5
$=>$, 5
#, 5
&, 5
$\rightsquigarrow$, 5
τ_{var}, 88
$\triangle_{=:}$, 8
$\vartheta_{autotactic}$, 8
ϑ_{pos}, 12
ϑ_{status}, 20
$\vartheta_{theorem}$, 12
$\vartheta_{universe}$, 8
in, 5
`:=`, 6
`=:`, 6
\, 5

acc type, 73
`add_def`, 10
`add_thm`, 9
`allequals`, 85
`apostroph`, 112
`appears`, 96
`appears_in`, 98
`apply`, 19
`arith`, 27
atom type, 32
`autotactic`, 11, 17, 19

`capital_letter`, 112
`cfringe`, 14
`cleandatabase`, 7
`complete`, 21, 22, 27
`compute`, 26, 106
`compute_list`, 104, 105
`compute_list_save`, 105
`compute_old`, 104
`compute_using`, 104
constructor rules, 30
`convertible`, 97
`convertlist`, 97
`copy`, 84, 112
`correct_tag`, 106
cproblem, 12
`create_def`, 3, 10
`create_thm`, 9
`cscreensize`, 14
`cshift`, 14
`current_def`, 11

`decide`, 26
`decideequal`, 85
`decimal`, 111
`decl`, 112
dependent function type, 56

dependent product type, 62
`derived`, 32
derived rule, 31
`diff`, 110
`digit`, 112
disjoint union type, 50
`display`, 15
`down`, 13
`dynvalue`, 6

`elim`, 26
`equal_combine`, 83
`equality`, 26
equality rules, 30, 31
equality type scheme, 75
`equals`, 85
`erase_def`, 11
`erasethm`, 11
`eval`, 101
`evalcond`, 104
`evallist`, 104
`ext`, 5
`extend`, 111
`extend_thin`, 109
`extract`, 17
`extractequal`, 85
`extractterm`, 9

`fairly_nonalphanum`, 88
`fold`, 24
`free`, 111
`freevar`, 111
`freevarinterm`, 96
`freevarsinterm`, 95
`fringe`, 14, 92
function type, 53

`generate_sons`, 14
`genint`, 111
`genvar`, 111
`goal`, 16

`hyp`, 27
`hyp_list`, 16
`hypothesis`, 17

`idtac`, 21
`idtaclist`, 25
`idtail`, 89
`illegal_rec_type`, 98
`imm_subterms`, 108
`increment`, 20
`info`, 1
int type, 41
`intro`, 26

less type scheme, 44
`level`, 111
`list`, 5
list type, 48
`load_thm`, 10

`makesubgoals`, 25
`mark`, 84
`markf`, 84
membership rules, 30, 31
membership type scheme, 75
`mod`, 5
`modify`, 96

natural numbers, 35
`new`, 28
`next`, 13
`noequal`, 30
`notused`, 111

`of`, 5
operator declarations, 5, 6, 21, 28
`or`, 21, 27

`pairing_of`, 93
`par`, 91
pless type scheme, 38
pnat type, 35

`pnat,pless,s_subterm,s_strip`, 97
`polish`, 17
`polishl`, 18
`pos`, 12
`prepare`, 92
product type, 60
prove, 21
`provelist`, 25
`proving`, 19
`pure`, 20, 27

`quote`, 92
quotient type, 65

`readfile`, 112
realisation rules, 30
recursive type, 70
`reduce`, 26
`refinement`, 17
refinement rules, 30, 31
`repeat`, 21, 22, 27
`replace_hyps`, 109
`rewrite`, 100
`rulename`, 32

`s`, 92
`save_def`, 11
`save_thm`, 10
`screensize`, 14
`select`, 12
selector rules, 30
`seq`, 27
`shift`, 14
`shyp`, 95
`simplify`, 26
`slist`, 95
`small_letter`, 112
`snapshot`, 15
`standardform`, 91
`status`, 16
`status0`, 17
`store`, 7
`stored`, 7
subset type, 67
`subst`, 26
`substituted`, 93
`substitutedlist`, 95
`syntax`, 86
`syntax0`, 86

`tab`, 112
`tactic`, 23, 24
`tag_for`, 107
`then`, 21
`thin`, 27
`thin_hyps`, 109
`toggle`, 112
`top`, 13
`treeassign`, 8
`try`, 21, 22, 27
`ttl1`, 88
`ttt`, 86
`tttl`, 88
`type`, 99
`type0`, 100

Unary type, 33
`underscore`, 112
`undo`, 20
`unfold`, 19
`universe`, 17, 19
universes, 77
`unroll`, 27
`unstore`, 7
`unstoreall`, 7
`up`, 13
user tactic, 23

`verify_tagging`, 106
`verify_unfold`, 106
void type, 34

`wdecl`, 89
`weight`, 91

`write_hyp`, 15
`writeclause`, 85
`writel`, 11
`writep`, 11
`writeterm`, 89
`writeterm0`, 89
`wterm`, 90
`wterml`, 91