The LEGO library – Version 1.3

LEGO Team <*lego@dcs.ed.ac.uk*>

June 15, 1998

Contents

1 Introduction

This report describes the current LEGO library, which was designed as a basis upon which users can develop their own work, and also some example proof developments using this library. An earlier version of the library was developed by Savi Maharaj for use by students being introduced to the LEGO system. We hope that this version will provide useful preliminaries for *all* users.

We have received contributions from: Thorsten Altenkirch; Petr Cizek; Adriana Compagnoni; Healf Goguen; Martin Hofmann; Claire Jones; Zhaohui Luo; Savi Maharaj; Conor McBride; James McKinna; Randy Pollack; Thomas Schreiber.

The LEGO library contains around 250 definitions and 400 theorems, and covers both the common datatypes such as natural numbers, list, booleans, products, sums, unit, empty and some more unusual ones such as integers, sigma types, quotient types, finite sets, vectors. New modules have been incorporated offering

- abstract data types
- a theory of relations
- a new theory of permutations
- a theory of sorted lists
- theorems on divisions

- theorems on exponentiation

The library exploits LEGO's module system so that loading a library file will automatically load the files on which it depends and two library files which depend on some common files will have the common files loaded only once. So users don't need to know anything about the dependency structure of the library, only where to find the theorems they want. For faster loading, the library also stores "compiled" versions of most files which can be accessed by using the `Make` command in LEGO.

2 Customising the Library

2.1 Universes

Given an inductive definition of a type I, one can exploit its generated elimination principle to define functions with domain I. The range of such a definition is limited by the universe of the elimination principle i.e., `Prop`, `Type(i)` or `Type`. Recall that `Prop` $\subsetneq$ `Type(i)` $\subsetneq$ `Type`. Hence, the theories of inductive definition vary according to the choice of universe. In particular, one cannot prove disjointness of constructors for any inductive definition which is eliminated over the kind `Prop`.

The environmental variable `LEGOPATH` controls the search path for LEGO modules. By default it accommodates elimination over `Type` and inductive types inhabiting `Type`. It is set to `.:$LIBDIR/lib_Type:$LIBDIR` where `LIBDIR` records the root directory of the library.

Users who prefer elimination over

`Prop` need to replace `lib_Type` by `lib_Prop` in their `LEGOPATH`. In this configuration, inductive types inhabit the kind `Prop`.

`Type(2)` need to replace `lib_Type` by `lib_TYPE` in their `LEGOPATH`. In this configuration, inductive types inhabit the kind `Type(0)`

More exotic configurations can be achieved by setting the variables `SET`, `TYPE`, `TYPE_minus1`, and `TYPE_minus2` in a module called `parameters` and ensure that

- their `LEGOPATH` considers their version of `parameters` before any such module in the library
- `LEGOPATH` has to include `lib_TYPE`.

Inductive types inhabit `SET` and the elimination rule eliminates over `TYPE`. Notice that some theorems in the library fail unless `TYPE_minus2` : `TYPE_minus1` : `TYPE`.

2.2 Equality

By default, the library provides Martin-Löf equality. Users who wish to work with a different notion of equality need to implement their own version of the module `lib_eq` and ensure that the `LEGOPATH` respects their choice . Provided this exports a relation `Eq`, a proof of its reflexivity and substitutivity `Eq_refl`, `Eq_subst`, the module `lib_eq_basics` will generate some theorems and configures the commands `Qrepl`, `Induction`, `Qnify` and the switch `Theorems`.

For convenience, a definition of Leibniz equality can be copied from the module `lib_eq_leibniz` and Christine Paulin-Mohring's inductive equality from `lib_eq_inductive`.

3 Conventions

In this document we will describe each module briefly and then give a list of the definitions made and theorems proved in each module. The syntax used in the module listings is the usual LEGO syntax. We restrict the documentation to the standard configuration. The types given are always the types produced when the library file is loaded. The notation "..." indicates omission of a proof term. Generally the only declarations which have been added to the context are the data-type constructors and elimination rules; all else is derived. The exceptions are in the definition of the quotient type and the axiom `true` $\neq$ `false` for booleans in the alternative library `lib_Prop` with elimination over `Prop`.

4 Providing logic and equality

4.1 Logic

The logic file contains impredicative encodings of "and", "or" "exists", etc. and their introduction and elimination rules, and definitions of some properties of relations (symmetry, reflexivity etc.). It is based on Randy Pollack's logic file. For the convenience of some of our users we have included definitions of "and" and "exists" which take up to 7 arguments.

```
** Module lib_logic Imports parameters
** Config Infix /\ left 3
** Config Infix \/ left 2
 cut = ... : {A,B|Prop}A->(A->B)->B
 trueProp = ... : Prop
 Id = ... : {T|SET}T->T
 PropId = ... : {A|Prop}A->A
 Contrapos = ... : {A,B|Prop}(A->B)->{X|Prop}(B->X)->A->X
 compose = ... : {T,S,U|SET}(S->U)->(T->S)->T->U
 permute = ... : {T,S,U|SET}(T->S->U)->S->T->U
 forall_elim = ... :
   {C|Prop}{T|SET}{P:T->Prop}({x:T}P x)->{e:T}((P e)->C)->C
 impl_elim = ... : {A,B,C|Prop}(A->B)->A->(B->C)->C
 and = ... : Prop->Prop->Prop
 or = ... : Prop->Prop->Prop
 or_elim = ... : {A,B,C|Prop}(A \/ B)->(A->C)->(B->C)->C
 pair = ... : {A,B|Prop}A->B->(A /\ B)
```

```
inl = ... : {A,B|Prop}A->(A \/ B)
inr = ... : {A,B|Prop}B->(A \/ B)
fst = ... : {A,B|Prop}(A /\ B)->A
snd = ... : {A,B|Prop}(A /\ B)->B
iff = ... : Prop->Prop->Prop
absurd = ... : Prop
absurd_elim = ... : absurd->{N:Prop}N
not = ... : Prop->Prop
All = ... : {T|SET}(T->Prop)->Prop
Ex = ... : {T|SET}(T->Prop)->Prop
ExIntro = ... : {T|SET}{wit:T}{P:T->Prop}(P wit)->Ex P
ExIn = ... :
  {T|SET}{wit:T}{P:T->Prop}(P wit)->{B:Prop}({t:T}(P t)->B)->B
ExElim = ... : {T|SET}{P|T->Prop}(Ex P)->{N|Prop}({t:T}(P t)->N)->N
ex = ... : {A|Prop}(A->Prop)->Prop
ex_intro = ... : {A|Prop}{P:A->Prop}{wit|A}(P wit)->ex P
witness = ... : {A|Prop}{P|A->Prop}(ex P)->A
Ex2 = ... : {T,S|SET}(T->S->Prop)->Prop
Ex2Intro = ... : {T,S|SET}{P:T->S->Prop}{t|T}{s|S}(P t s)->Ex2 P
Ex2Elim = ... :
  {T,S|SET}{P|T->S->Prop}(Ex2 P)->{N|Prop}({t|T}{s|S}(P t s)->N)->N
Ex2In = ... : {T,S|SET}{t:T}{s:S}{P:T->S->Prop}(P t s)->Ex2 P
Ex3 = ... : {T,S,U|SET}(T->S->U->Prop)->Prop
Ex3In = ... :
  {T,S,U|SET}{t:T}{s:S}{u:U}{P:T->S->U->Prop}(P t s u)->Ex3 P
Ex3Elim = ... :
  {T,S,U|SET}{P|T->S->U->Prop}(Ex3 P)->{N|Prop}
  ({t|T}{s|S}{u|U}(P t s u)->N)->N
Ex4 = ... : {T,S,U,V|SET}(T->S->U->V->Prop)->Prop
Ex4In = ... :
  {T,S,U,V|SET}{t:T}{s:S}{u:U}{v:V}{P:T->S->U->V->Prop}(P t s u v)->
  Ex4 P
Ex4Elim = ... :
  {T,S,U,V|SET}{P|T->S->U->V->Prop}(Ex4 P)->{N|Prop}
  ({t|T}{s|S}{u|U}{v|V}(P t s u v)->N)->N
Ex5 = ... : {T,S,U,V,W|SET}(T->S->U->V->W->Prop)->Prop
Ex5In = ... :
  {T,S,U,V,W|SET}{t:T}{s:S}{u:U}{v:V}{w:W}{P:T->S->U->V->W->Prop}
  (P t s u v w)->Ex5 P
Ex5Elim = ... :
  {T,S,U,V,W|SET}{P|T->S->U->V->W->Prop}(Ex5 P)->{N|Prop}
  ({t|T}{s|S}{u|U}{v|V}{w|W}(P t s u v w)->N)->N
Ex6 = ... : {T,S,U,V,W,X|SET}(T->S->U->V->W->X->Prop)->Prop
```

```
Ex6In = ... :
  {T,S,U,V,W,X|SET}{t:T}{s:S}{u:U}{v:V}{w:W}{x:X}
  {P:T->S->U->V->W->X->Prop}(P t s u v w x)->Ex6 P
Ex6Elim = ... :
  {T,S,U,V,W,X|SET}{P|T->S->U->V->W->X->Prop}(Ex6 P)->{N|Prop}
  ({t|T}{s|S}{u|U}{v|V}{w|W}{x|X}(P t s u v w x)->N)->N
and3 = ... : Prop->Prop->Prop->Prop
pair3 = ... : {A,B,C|Prop}A->B->C->and3 A B C
and3_out1 = ... : {A,B,C|Prop}(and3 A B C)->A
and3_out2 = ... : {A,B,C|Prop}(and3 A B C)->B
and3_out3 = ... : {A,B,C|Prop}(and3 A B C)->C
and4 = ... : Prop->Prop->Prop->Prop->Prop
pair4 = ... : {A,B,C,D|Prop}A->B->C->D->and4 A B C D
and4_out1 = ... : {A,B,C,D|Prop}(and4 A B C D)->A
and4_out2 = ... : {A,B,C,D|Prop}(and4 A B C D)->B
and4_out3 = ... : {A,B,C,D|Prop}(and4 A B C D)->C
and4_out4 = ... : {A,B,C,D|Prop}(and4 A B C D)->D
and5 = ... : Prop->Prop->Prop->Prop->Prop->Prop
pair5 = ... : {A,B,C,D,E|Prop}A->B->C->D->E->and5 A B C D E
and5_out1 = ... : {A,B,C,D,E|Prop}(and5 A B C D E)->A
and5_out2 = ... : {A,B,C,D,E|Prop}(and5 A B C D E)->B
and5_out3 = ... : {A,B,C,D,E|Prop}(and5 A B C D E)->C
and5_out4 = ... : {A,B,C,D,E|Prop}(and5 A B C D E)->D
and5_out5 = ... : {A,B,C,D,E|Prop}(and5 A B C D E)->E
and6 = ... : Prop->Prop->Prop->Prop->Prop->Prop->Prop
pair6 = ... : {A,B,C,D,E,F|Prop}A->B->C->D->E->F->and6 A B C D E F
and6_out1 = ... : {A,B,C,D,E,F|Prop}(and6 A B C D E F)->A
and6_out2 = ... : {A,B,C,D,E,F|Prop}(and6 A B C D E F)->B
and6_out3 = ... : {A,B,C,D,E,F|Prop}(and6 A B C D E F)->C
and6_out4 = ... : {A,B,C,D,E,F|Prop}(and6 A B C D E F)->D
and6_out5 = ... : {A,B,C,D,E,F|Prop}(and6 A B C D E F)->E
and6_out6 = ... : {A,B,C,D,E,F|Prop}(and6 A B C D E F)->F
and7 = ... : Prop->Prop->Prop->Prop->Prop->Prop->Prop->Prop
pair7 = ... :
  {A,B,C,D,E,F,G|Prop}A->B->C->D->E->F->G->and7 A B C D E F G
and7_out1 = ... : {A,B,C,D,E,F,G|Prop}(and7 A B C D E F G)->A
and7_out2 = ... : {A,B,C,D,E,F,G|Prop}(and7 A B C D E F G)->B
and7_out3 = ... : {A,B,C,D,E,F,G|Prop}(and7 A B C D E F G)->C
and7_out4 = ... : {A,B,C,D,E,F,G|Prop}(and7 A B C D E F G)->D
and7_out5 = ... : {A,B,C,D,E,F,G|Prop}(and7 A B C D E F G)->E
and7_out6 = ... : {A,B,C,D,E,F,G|Prop}(and7 A B C D E F G)->F
and7_out7 = ... : {A,B,C,D,E,F,G|Prop}(and7 A B C D E F G)->G
iff3 = ... : Prop->Prop->Prop->Prop
```

```
or3 = ... : Prop->Prop->Prop->Prop
or3_in1 = ... : {A,B,C|Prop}A->or3 A B C
or3_in2 = ... : {A,B,C|Prop}B->or3 A B C
or3_in3 = ... : {A,B,C|Prop}C->or3 A B C
or3_elim = ... : {A,B,C,X|Prop}(or3 A B C)->(A->X)->(B->X)->(C->X)->X
refl = ... : {T|SET}(T->T->Prop)->Prop
sym = ... : {T|SET}(T->T->Prop)->Prop
trans = ... : {T|SET}(T->T->Prop)->Prop
Equivalence = ... : {T|SET}(T->T->Prop)->Prop
respect = ... : {T,S|SET}(T->S)->({X|SET}X->X->Prop)->Prop
respect2 = ... : {T,S,U|SET}(T->U->S)->({X|SET}X->X->Prop)->Prop
```

4.2 Equality

This section describes the directory `lib_equality` in which various definitions of equality are kept.

4.2.1 Martin-Löf's Inductive Equality

This library file gives the definition and rules for Martin-Löf's inductive equality. We cannot use LEGO's inductive type mechanism for this definition because the type of the constructor `Eqr` does not conform to Luo's schema for inductive types, so the definition is done by hand.

```
 ** Module lib_eq Imports lib_logic
 $Eq : {t|Type}t->t->Prop
 $Eqr : {t|Type}{x:t}Eq x x
 $Eq_elim :
    {t|Type}{P:{x,y:t}(Eq x y)->Type}({x:t}P x x (Eqr x))->{x,y:t}
    {p:Eq x y}P x y p
[[t:Type][P:{x,y:t}(Eq x y)->Type][h:{x:t}P x x (Eqr x)][x:t]
    Eq_elim P h x x (Eqr x)  ==>  h x]

  Eq_refl = ... : {t|Type}refl (Eq|t)
  Eq_subst = ... : {t|Type}{m,n|t}(Eq m n)->{P:t->Type}(P m)->P n
  trivType = ... : Type
  emptyType = ... : Type
 ** Config Theorems trivType emptyType Eq_subst
```

4.2.2 Leibniz Equality

This module contains the definition and rules for manipulating Leibniz (impredicative) equality. This file is based on the definitions in Randy Pollack's logic file.

```
** Module lib_eq_leibniz Imports lib_logic
 Eq = ... : {T|Type}T->T->Prop
 Eq_refl = ... : {T|Type}refl (Eq|T)
 Eq_subst = ... : {T|Type}{x,y|T}(Eq x y)->{P:T->Prop}(P x)->P y
```

4.2.3 Paulin-Mohring's Inductive Equality

The next library equality file gives the definition and rules for Christine Paulin-Mohring's inductive equality.

```
 ** Module lib_eq_inductive Imports lib_logic
 $Eq : {t|Type}t->t->Prop
 $Eqr : {t|Type}{x:t}Eq x x
 $Eq_elim :
    {t|Type}{x:t}{C_Eq:{x1|t}(Eq x x1)->Type}(C_Eq (Eqr x))->{x1|t}
    {z:Eq x x1}C_Eq z
[[t|Type][x:t][C_Eq:{x1|t}(Eq x x1)->Type][f_Eqr:C_Eq (Eqr x)]
    Eq_elim x C_Eq f_Eqr (Eqr x)  ==>  f_Eqr]

  Eq_refl = ... : {t|Type}refl (Eq|t)
  Eq_subst = ... : {t|Type}{m,n|t}(Eq m n)->{P:t->Type}(P m)->P n
```

4.2.4 Altenkirch-Streicher Elimination Rule

This file contains an extra elimination rule for Martin-Löf's inductive equality proposed by Thorsten Altenkirch and Thomas Streicher. This new rule is used to prove the proposition `Eq_unique` below which allows one to prove e.g. that equality in inductive sigma types respects second projection. This rule has recently been shown not to be derivable from the usual elimination rule for inductive equality (Hofmann & Streicher 1994).

```
 ** Module lib_AS_eq Imports lib_eq
 $Eq_elim' :
    {A|Type}{C:{a|A}(Eq a a)->Type}({a:A}C (Eq_refl a))->{a|A}{p:Eq a a}
    C p
  Eq_unique = ... : {A|Type}{a,b:A}{p,q:Eq a b}Eq p q
```

4.2.5 Theorems about Equality

All of the above notions of equality are reflexive and substitutive. We can thus derive symmetry and transitivity. Furthermore, equality respects function applications i.e.,

$$\frac{a = b}{f(a) = f(b)} \qquad \text{(Eq\_resp)}$$

$$\frac{a_1 = b_1 \quad a_2 = b_2}{f(a_1)(a_2) = f(b_1)(b_2)} \qquad \text{(Eq\_resp2)}$$

```
** Module lib_eq_basics Imports lib_eq
 injective = ... : {S|Type}{T|Type}(S->T)->Prop
 Eq_sym = ... : {t|Type}sym (Eq|t)
 Eq_trans = ... : {t|Type}trans (Eq|t)
 Eq_resp = ... : {A|Type}{B|Type}{f:A->B}respect f Eq
** Config Equality Eq Eq_refl Eq_subst
** Config Qrepl Eq Eq_subst Eq_sym
** Config Qrepl Eq Eq_subst Eq_sym
 Eq_resp2 = ... : {A|Type}{B|Type}{C|Type}{r:A->B->C}respect2 r Eq
```

4.3 Setoids

Setoids are algebraic structures characterised by a domain and an equality relation.

```
 ** Module lib_setoid Imports parameters
 $Setoid : Type(Setoid)
 $make_Setoid : {dom:Type}(dom->dom->Prop)->Setoid
 $Setoid_elim :
    {C_Setoid:Setoid->TYPE}
    ({dom:Type}{eq1:dom->dom->Prop}C_Setoid (make_Setoid dom eq1))->
    {z:Setoid}C_Setoid z
[[C_Setoid:Setoid->TYPE]
 [f_make_Setoid:{dom:Type}{eq1:dom->dom->Prop}
  C_Setoid (make_Setoid dom eq1)][dom:Type][eq1:dom->dom->Prop]
    Setoid_elim C_Setoid f_make_Setoid (make_Setoid dom eq1)  ==>
    f_make_Setoid dom eq1]

  dom = ... : Setoid->Type
  eq1 = ... : {z:Setoid}(dom z)->(dom z)->Prop
  Setoid_eq = ... : {S|Setoid}(dom S)->(dom S)->Prop
```

5 The Empty Type

The empty type is defined in the directory `lib_empty`.

```
 ** Module lib_empty Imports parameters
```

```
 $empty : Type(empty)
 $empty_elim : {C_empty:empty->TYPE}{z:empty}C_empty z
  empty_rec = ... : {t|Type}empty->t
  empty_ind = ... : {P:empty->Prop}{z:empty}P z
```

6 The Unit Type

The one element type is defined in the directory `lib_unit`.

```
 ** Module lib_unit Imports lib_eq_basics
 $unit : Type(unit)
 $void : unit
 $unit_elim : {C_unit:unit->Type}(C_unit void)->{z:unit}C_unit z
[[C_unit:unit->Type][f_void:C_unit void]
    unit_elim C_unit f_void void  ==>  f_void]

  unit_rec = ... : {t|Type}t->unit->t
  unit_ind = ... : {P:unit->Prop}(P void)->{u:unit}P u
  is_void = ... : unit->Prop
  all_units_void = ... : {x:unit}is_void x
  all_units_same = ... : {x,y:unit}Eq x y
```

7 The Booleans

There are three library files in the directory `lib_bool`.

7.1 Boolean Definition

The file `lib_bool` gives the basic definition of the boolean type with its recursor and induction principle and propositional functions `is_true` and `is_false` which are used in the two characterisation theorems that true is not equal to false and that any boolean is either true or false.

```
 ** Module lib_bool Imports lib_eq_basics
 $bool : Type(bool)
 $true : bool
 $false : bool
 $bool_elim :
    {C_bool:bool->TYPE}(C_bool true)->(C_bool false)->{z:bool}C_bool z
[[C_bool:bool->TYPE][f_true:C_bool true][f_false:C_bool false]
    bool_elim C_bool f_true f_false true  ==>  f_true
 || bool_elim C_bool f_true f_false false  ==>  f_false]
```

```
 bool_is_true = ... : bool->Type
 bool_is_false = ... : bool->Type
 bool_true_injective = ... : (Eq true true)->{P|Type}P->P
 bool_false_injective = ... : (Eq false false)->{P|Type}P->P
 bool_rec = ... : {T|Type}T->T->bool->T
 bool_ind = ... : {P:bool->Prop}(P true)->(P false)->{b:bool}P b
 bool_double_elim = ... :
   {T:bool->bool->Type}(T true true)->(T true false)->(T false true)->
   (T false false)->{x,y:bool}T x y
 is_true = ... : bool->Prop
 is_false = ... : bool->Prop
 true_not_false = ... : not (Eq true false)
 true_or_false = ... : {b:bool}(is_true b \/ is_false b)
```

7.2 Boolean Functions

The file of boolean functions defines the boolean counterparts to the usual propositional connectives, if, and, or, implies etc. We also define the boolean-valued equality function on booleans.

```
** Module lib_bool_funs Imports lib_bool
 if = ... : {t|Type}bool->t->t->t
 andalso = ... : bool->bool->bool
 orelse = ... : bool->bool->bool
 inv = ... : bool->bool
 implies = ... : bool->bool->bool
 bool_eq = ... : bool->bool->bool
```

7.3 Boolean Theorems

This file contains various theorems about the above boolean functions, mostly proving that they have the expected meanings.

```
** Module lib_bool_thms Imports lib_bool_funs
 is_true_not_is_false = ... :
   {b:bool}iff (is_true b) (not (is_false b))
 bool_contrapos = ... :
   {a,b|bool}((is_true a)->is_true b)->(is_false b)->is_false a
 bool_contradiction = ... : {b|bool}(is_true b)->(is_false b)->absurd
 if_absorbs = ... : {b:bool}{t|Type}{x:t}Eq (if b x x) x
 if_character = ... :
   {t|Type}{x,y,z|t}{b|bool}(Eq (if b x y) z)->(Eq x z \/ Eq y z)
```

```
Eq_resp_if = ... :
  {t|Type}{x,y,x',y'|t}(Eq x x')->(Eq y y')->{b:bool}
  Eq (if b x y) (if b x' y')
if_inside = ... :
  {s|Type}{t|Type}{f:s->t}{x,y:s}{b:bool}
  Eq (if b (f x) (f y)) (f (if b x y))
middle_four_if = ... :
  {A|Type}{a,b,c,d:A}{x,y:bool}
  Eq (if x (if y a b) (if y c d)) (if y (if x a c) (if x b d))
reduce_if_left = ... :
  {A|Type}{a,b,c:A}{x:bool}Eq (if x (if x a b) c) (if x a c)
reduce_if_right = ... :
  {A|Type}{a,b,c:A}{x:bool}Eq (if x a (if x b c)) (if x a c)
if_prop = ... : {C|Type}{b:bool}((Eq b true)->C)->((Eq b false)->C)->C
andalso_character = ... :
  {a,b:bool}iff (is_true (andalso a b)) (is_true a /\ is_true b)
andalso_absorbs = ... : {b:bool}Eq (andalso b b) b
andalso_commutes = ... : {b,c:bool}Eq (andalso b c) (andalso c b)
andalso_false = ... : {b:bool}Eq (andalso b false) false
andalso_true = ... : {b:bool}Eq (andalso b true) b
inv_character = ... : {b:bool}iff (is_true (inv b)) (not (is_true b))
inv_character' = ... : {a,b:bool}iff (Eq (inv a) b) (Eq a (inv b))
inv_idemp = ... : {b:bool}Eq (inv (inv b)) b
inv_is_true_is_false = ... :
  {n:bool}iff (is_true (inv n)) (is_false n)
inv_is_false_is_true = ... :
  {n:bool}iff (is_false (inv n)) (is_true n)
orelse_character = ... :
  {a,b:bool}iff (is_true (orelse a b)) (is_true a \/ is_true b)
orelse_absorbs = ... : {b:bool}Eq (orelse b b) b
orelse_assoc = ... :
  {a,b,c:bool}Eq (orelse (orelse a b) c) (orelse a (orelse b c))
orelse_commutes = ... : {a,b:bool}Eq (orelse a b) (orelse b a)
orelse_false = ... : {b:bool}Eq (orelse b false) b
orelse_true = ... : {b:bool}Eq (orelse b true) true
deMorgan = ... :
  {a,b:bool}iff (is_false (orelse a b)) (is_false a /\ is_false b)
deMorgan' = ... :
  {a,b:bool}iff (is_false (andalso a b)) (is_false a \/ is_false b)
implies_character = ... :
  {a,b:bool}iff (is_true (implies a b)) ((is_true a)->is_true b)
implies_false = ... : {n:bool}is_true (implies false n)
```

8 Product Types

There is one library file in the directory `lib_prod` with the definition of (non-dependent) cross-product. As well as the basic definitions there are also theorems giving the η-rule for pairs and the fact that pairing is extensional.

```
 ** Module lib_prod Imports lib_eq_basics
 $prod : (Type)->(Type)->Type(prod)
 $pair1 : {A:Type}{B:Type}A->B->prod A B
 $prod_elim :
    {A:Type}{B:Type}{C_prod:(prod A B)->Type}
    ({a:A}{b:B}C_prod (pair1 A B a b))->{z:prod A B}C_prod z
[[A:Type][B:Type][C_prod:(prod A B)->Type]
 [f_pair1:{a:A}{b:B}C_prod (pair1 A B a b)][a:A][b:B]
    prod_elim A B C_prod f_pair1 (pair1 A B a b)  ==>  f_pair1 a b]

  prod_is_pair1 = ... : {A:Type}{B:Type}(prod A B)->Type
  prod_pair1_injective = ... :
    {A:Type}{B:Type}{ix0,iy0|A}{ix1,iy1|B}
    (Eq (pair1 A B ix0 ix1) (pair1 A B iy0 iy1))->{P|Type}
    ((Eq ix0 iy0)->(Eq ix1 iy1)->P)->P
  Pair = ... : {A|Type}{B|Type}A->B->prod A B
  prod_rec = ... : {s|Type}{t|Type}{u|Type}(s->t->u)->(prod s t)->u
  prod_ind = ... :
    {s|Type}{t|Type}{P:(prod s t)->Prop}({a:s}{b:t}P (Pair a b))->
    {p:prod s t}P p
  Fst = ... : {s|Type}{t|Type}(prod s t)->s
  Snd = ... : {s|Type}{t|Type}(prod s t)->t
  prod_eta = ... :
    {s|Type}{t|Type}{p:prod s t}Eq p (Pair (Fst p) (Snd p))
  prod_ext = ... :
    {s|Type}{t|Type}{p,q:prod s t}(Eq (Fst p) (Fst q))->
    (Eq (Snd p) (Snd q))->Eq p q
```

9 Sum Types

In the directory `lib_sum` there is a single file defining the sum type with constructors `in1` and `in2`. Also defined is the case function (essentially the iterator for sum).

```
 ** Module lib_sum Imports lib_eq_basics
 $sum : (Type)->(Type)->Type(sum)
 $in1_exp : {A:Type}{B:Type}A->sum A B
 $in2_exp : {A:Type}{B:Type}B->sum A B
```

```
 $sum_elim :
    {A:Type}{B:Type}{C_sum:(sum A B)->Type}
    ({x2:A}C_sum (in1_exp A B x2))->({x1:B}C_sum (in2_exp A B x1))->
    {z:sum A B}C_sum z
  sum_double_elim = ... :
    {A:Type}{B:Type}{T_sum:(sum A B)->(sum A B)->Type}
    ({x2,x2':A}T_sum (in1_exp A B x2) (in1_exp A B x2'))->
    ({x2:A}{x1':B}T_sum (in1_exp A B x2) (in2_exp A B x1'))->
    ({x1:B}{x2':A}T_sum (in2_exp A B x1) (in1_exp A B x2'))->
    ({x1,x1':B}T_sum (in2_exp A B x1) (in2_exp A B x1'))->{z,z':sum A B}
    T_sum z z'
[[A:Type][B:Type][C_sum:(sum A B)->Type]
 [f_in1_exp:{x2:A}C_sum (in1_exp A B x2)]
 [f_in2_exp:{x1:B}C_sum (in2_exp A B x1)][x2:A][x1:B]
    sum_elim A B C_sum f_in1_exp f_in2_exp (in1_exp A B x2)  ==>
    f_in1_exp x2
 || sum_elim A B C_sum f_in1_exp f_in2_exp (in2_exp A B x1)  ==>
    f_in2_exp x1]

  in1 = ... : {s|Type}{t|Type}s->sum s t
  in2 = ... : {s|Type}{t|Type}t->sum s t
  case = ... : {s|Type}{t|Type}{u|Type}(s->u)->(t->u)->(sum s t)->u
  sum_ind = ... :
    {s|Type}{t|Type}{P:(sum s t)->Prop}({x:s}P (in1 x))->
    ({y:t}P (in2 y))->{z:sum s t}P z
  is_in1 = ... : {s|Type}{t|Type}(sum s t)->Prop
  is_in2 = ... : {s|Type}{t|Type}(sum s t)->Prop
  in1_or_in2 = ... : {s|Type}{t|Type}{x:sum s t}(is_in1 x \/ is_in2 x)
  in1_not_in2 = ... : {s|Type}{t|Type}{x|s}not (is_in2 (in1|s|t x))
  in2_not_in1 = ... : {s|Type}{t|Type}{x|t}not (is_in1 (in2|s|t x))
  in1_inj = ... :
    {s|Type}{t|Type}{x,y:s}(Eq (in1|s|t x) (in1|s|t y))->Eq x y
  in2_inj = ... :
    {s|Type}{t|Type}{x,y:t}(Eq (in2|s|t x) (in2|s|t y))->Eq x y
```

10 Inductive Sigma

In the directory `lib_sigma` we define the inductive sigma type, with its projections, a proof that the first projection is injective and the eta-rule for the dependent pairs. Products can then be defined as a non-dependent sigma type.

```
 ** Module lib_sigma Imports lib_eq_basics
```

```
 $sigma : {A|Type}(A->Type)->Type(sigma)
 $dep_pair : {A|Type}{B|A->Type}{a:A}(B a)->sigma|A|B
 $sigma_elim :
    {A|Type}{B|A->Type}{C_sigma:(sigma|A|B)->Type}
    ({a:A}{b:B a}C_sigma (dep_pair a b))->{z:sigma|A|B}C_sigma z
[[A|Type][B|A->Type][C_sigma:(sigma|A|B)->Type]
 [f_dep_pair:{a:A}{b:B a}C_sigma (dep_pair a b)][a:A][b:B a]
    sigma_elim C_sigma f_dep_pair (dep_pair a b)  ==>  f_dep_pair a b]

  sigma_rec = ... :
    {A|Type}{B|A->Type}{T|Type}({a:A}(B a)->T)->(sigma|A|B)->T
  sigma_ind = ... :
    {A|Type}{B|A->Type}{P:(sigma|A|B)->Prop}
    ({a:A}{b:B a}P (dep_pair a b))->{z:sigma|A|B}P z
  sig_pi1 = ... : {A|Type}{B|A->Type}(sigma|A|B)->A
  sig_pi2 = ... : {A|Type}{B|A->Type}{z:sigma|A|B}B (sig_pi1 z)
  dep_pair_eta = ... :
    {A|Type}{B|A->Type}{x:sigma|A|B}
    Eq x (dep_pair (sig_pi1 x) (sig_pi2 x))
  sig_pi1_inj = ... :
    {A|Type}{B|A->Type}{x,y:sigma|A|B}(Eq x y)->
    Eq (sig_pi1 x) (sig_pi1 y)
  Sigma = ... : {A:Type}(A->Type)->Type(sigma)
  Prod = ... : (Type)->(Type)->Type(sigma)
  Pr = ... : {A|Type}{B|Type}A->B->sigma|A|([_:A]B)
```

11 The Natural Numbers

The directory `lib_nat` is the biggest section of the library so far. It contains a number of files, each concerning a particular function or set of functions, so users can load just the theorems relating to the functions they want. Some files contain only a few theorems in their own right but load many theorems from other files on which they depend. An example of this is the file `lib_nat_rels` which loads all the files about relations on the natural numbers and also most of the files with theorems about the algebraic structure of nat.

11.1 Natural Numbers: Basic Definitions

The basic definitions are held in the file `lib_nat`. We find it useful to define the double iterator and double induction, and the constants `one` to `ten`. This file also defines the algebraic functions addition, multiplication and exponentiation. We also define `pred` (predecessor) in one step from this we define truncated subtraction (`minus`).

```
** Module lib_nat Imports lib_bool
$nat : Type(nat)
$zero : nat
$suc : nat->nat
$nat_elim :
   {C_nat:nat->Type}(C_nat zero)->
   ({x1:nat}(C_nat x1)->C_nat (suc x1))->{z:nat}C_nat z
[[C_nat:nat->Type][f_zero:C_nat zero]
 [f_suc:{x1:nat}(C_nat x1)->C_nat (suc x1)][x1:nat]
   nat_elim C_nat f_zero f_suc zero  ==>  f_zero
|| nat_elim C_nat f_zero f_suc (suc x1)  ==>
   f_suc x1 (nat_elim C_nat f_zero f_suc x1)]

 nat_is_zero = ... : nat->Type
 nat_is_suc = ... : nat->Type
 nat_zero_injective = ... : (Eq zero zero)->{P|Type}P->P
 nat_suc_injective = ... :
   {ix0,iy0|nat}(Eq (suc ix0) (suc iy0))->{P|Type}((Eq ix0 iy0)->P)->P
 nat_rec = ... : {t|Type}t->(nat->t->t)->nat->t
 nat_iter = ... : {t|Type}t->(t->t)->nat->t
 nat_ind = ... :
   {P:nat->Prop}(P zero)->({n:nat}(P n)->P (suc n))->{n:nat}P n
 nat_double_elim = ... :
   {T:nat->nat->Type}(T zero zero)->
   ({n:nat}(T zero n)->T zero (suc n))->
   ({m:nat}({n:nat}T m n)->T (suc m) zero)->
   ({m:nat}({n:nat}T m n)->{n:nat}(T (suc m) n)->T (suc m) (suc n))->
   {m,n:nat}T m n
 nat_diagonal_iter = ... :
   {C|Type}(nat->C)->(nat->C)->(C->C)->nat->nat->C
 nat_diagonal_ind = ... :
   {phi:nat->nat->Prop}({m:nat}phi zero m)->({n:nat}phi (suc n) zero)->
   ({n,m:nat}(phi n m)->phi (suc n) (suc m))->{n,m:nat}phi n m
 nat_d_rec_for_int = ... :
   {C:nat->nat->Type}(C zero zero)->({m,n:nat}(C m n)->C (suc m) n)->
   ({m,n:nat}(C m n)->C m (suc n))->{m,n:nat}C m n
 is_suc = ... : nat->Prop
 is_zero = ... : nat->Prop
 zero_not_suc = ... : {n:nat}not (Eq zero (suc n))
 suc_not_zero = ... : not (is_suc zero)
 suc_n_not_n = ... : {n:nat}not (Eq (suc n) n)
** Config Qnify nat suc_n_not_n
 zero_or_suc = ... : {n:nat}(Eq n zero \/ is_suc n)
```

```
one = ... : nat
two = ... : nat
three = ... : nat
four = ... : nat
five = ... : nat
six = ... : nat
seven = ... : nat
eight = ... : nat
nine = ... : nat
ten = ... : nat
plus = ... : nat->nat->nat
times = ... : nat->nat->nat
exp = ... : nat->nat->nat
pred = ... : nat->nat
minus = ... : nat->nat->nat
ackerman = ... : nat->nat->nat
if_zero = ... : {t|Type}nat->t->t->t
fact = ... : nat->nat
zero_not_one = ... : not (Eq zero one)
```

11.2 Iteration Principles

```
** Module lib_nat_iter Imports lib_nat lib_bool_funs
 nat_iter_suc = ... :
   {t|Type}{z:t}{s:t->t}{n:nat}
   Eq (nat_iter z s (suc n)) (nat_iter (s z) s n)
 while_cond1 = ... :
   {f:nat->bool}[lll=nat_iter zero ([a:nat]if (f (suc a)) (suc a) a)]
   (is_true (f zero))->{n:nat}is_true (f (lll n))
 while_cond2 = ... :
   {f:nat->bool}[lll=nat_iter zero ([a:nat]if (f (suc a)) (suc a) a)]
   {n:nat}(Eq (lll n) n \/ is_false (f (suc (lll n))))
```

11.3 Induction Principles

This short file gives some induction principles which deal with the recursor and iterator on the natural numbers. For their use in inductive proofs using deliverables see (McKinna 1992).

```
** Module lib_more_induction Imports lib_nat
 nat_iter_induction = ... :
   [Pred=[a:Type]a->Prop]{gamma|Type}{z|gamma}{s|gamma->gamma}
   {phi:Pred gamma}(phi z)->({g:gamma}(phi g)->phi (s g))->
```

```
  [rec=nat_iter z s]{n:nat}phi (rec n)
nat_rec_induction = ... :
  [Rel=[a:Type][b:Type]a->b->Prop]{gamma|Type}{z|gamma}
  {s|nat->gamma->gamma}{phi:Rel nat gamma}(phi zero z)->
  ({m:nat}{g:gamma}(phi m g)->phi (suc m) (s m g))->[rec=nat_rec z s]
  {n:nat}phi n (rec n)
nat_iter_induction_rev = ... :
  [Pred=[a:Type]a->Prop]{gamma|Type}{z|gamma}{s|gamma->gamma}
  {phi:Pred gamma}{n:nat}(phi (nat_iter z s n))->
  ({g:gamma}(phi (s g))->phi g)->phi z
```

11.4 Theorems about Successor

This short file gives the theorems which rely only on successor and predecessor functions.

```
** Module lib_nat_suc_thms Imports lib_nat
 Eq_resp_suc = ... : {m,n|nat}(Eq m n)->Eq (suc m) (suc n)
 Eq_resp_pred = ... : {m,n|nat}(Eq m n)->Eq (pred m) (pred n)
 suc_injective = ... : {m,n|nat}(Eq (suc m) (suc n))->Eq m n
 n_not_suc_n = ... : {n:nat}not (Eq n (suc n))
 suc_pred = ... : {n|nat}(not (Eq n zero))->Eq n (suc (pred n))
```

11.5 Theorems about Minus

Another small file gives a few theorems about truncated subtraction (`minus`).

```
** Module lib_nat_minus_thms Imports lib_nat_plus_thms
 minus_suc_pred = ... :
   {n,m:nat}Eq (minus n (suc m)) (minus (pred n) m)
 minus_resp_suc = ... : {m,n:nat}Eq (minus (suc m) (suc n)) (minus m n)
 n_minus_n_Eq_zero = ... : {n:nat}Eq (minus n n) zero
 zero_minus_n_Eq_zero = ... : {n:nat}Eq (minus zero n) zero
 minus_commutes_suc_l = ... :
   {m,n,t|nat}(Eq (minus n m) (suc t))->
   Eq (minus (suc n) m) (suc (minus n m))
 minus_commutes_suc_r = ... :
   {m,n,t|nat}(Eq (minus n m) (suc t))->Eq (minus n (suc m)) t
 minus_monotone_l = ... :
   {l:nat}{m,n,t|nat}(Eq (minus n m) (suc t))->
   Eq (minus (plus n l) m) (suc (plus t l))
 minus_inv_plus = ... : {m,n:nat}Eq n (minus (plus m n) m)
 nonzero_inv_sp = ... : {x:nat}(not (Eq x zero))->Eq (suc (pred x)) x
```

11.6 Theorems about Plus

Theorems involving only the function plus are given in this file below. Note the theorem `middle_four_plus`—we find this and similar theorems very useful for manipulating expressions involving commutative, associative operators. It is possible to prove the general case of this theorem and derive this and other instances of it from it as desired. We may choose to do this in a future version of the library.

```
** Module lib_nat_plus_thms Imports lib_nat_suc_thms
 plus_zero = ... : {n:nat}Eq (plus n zero) n
 plus_suc = ... : {m,n:nat}Eq (plus m (suc n)) (suc (plus m n))
 plus_suc_lemma = ... : {x,t:nat}not (Eq (plus x (suc t)) x)
 plus_assoc = ... :
   {k,m,n:nat}Eq (plus (plus k m) n) (plus k (plus m n))
 plus_commutes = ... : {m,n:nat}Eq (plus m n) (plus n m)
 plus_strict = ... :
   {n,m|nat}(Eq (plus n m) zero)->(Eq n zero /\ Eq m zero)
 middle_four_plus = ... :
   {i,j,k,l|nat}
   Eq (plus (plus i k) (plus j l)) (plus (plus i j) (plus k l))
 one_plus_lemma = ... :
   {m,n|nat}(Eq one (plus m n))->
   ((Eq one m /\ Eq zero n) \/ (Eq zero m /\ Eq one n))
 cancel_plus = ... : {i,j,k:nat}(Eq (plus i j) (plus i k))->Eq j k
 plus_subst_l = ... : {l,n,m|nat}(Eq l n)->Eq (plus l m) (plus n m)
 plus_subst_r = ... : {l,n,m|nat}(Eq l n)->Eq (plus m l) (plus m n)
```

11.7 Theorems about multiplication

In this file are theorems about multiplication, including those also involving addition e.g. the distributivity laws.

```
** Module lib_nat_times_thms Imports lib_nat_plus_thms
 times_plus_dist_r = ... :
   {i,j,k|nat}Eq (times (plus i j) k) (plus (times i k) (times j k))
 times_plus_dist_l = ... :
   {i,j,k|nat}Eq (times i (plus j k)) (plus (times i j) (times i k))
 times_assoc = ... :
   {i,j,k|nat}Eq (times (times i j) k) (times i (times j k))
 times_zero = ... : {n|nat}Eq zero (times n zero)
 times_one = ... : {n|nat}Eq n (times one n)
 zero_times_lemma = ... :
   {m,n|nat}(Eq zero (times m n))->(Eq zero m \/ Eq zero n)
 one_times_lemma = ... :
```

```
  {m,n|nat}(Eq one (times m n))->(Eq one m /\ Eq one n)
 times_commutes = ... : {m,n|nat}Eq (times m n) (times n m)
 times_cancel_right = ... :
   {a,x,y:nat}(Eq (times x (suc a)) (times y (suc a)))->Eq x y
 times_two = ... : {n:nat}Eq (times two n) (plus n n)
```

11.8 Square Roots

In this file are theorems about squares and square roots.

```
** Module lib_nat_sqrt Imports lib_nat_iter lib_nat_rels
 square = ... : nat->nat
 sqrt = ... : nat->nat
 sqrt_cor = ... :
   {n:nat}(Le (square (sqrt n)) n /\ Lt n (square (suc (sqrt n))))
 sqrt_small = ... : {n:nat}(Lt one n)->Lt (sqrt n) n
```

11.9 Order relations

An important class of functions on the natural numbers are the order relations, strict less-than and less than or equal. There are many ways to define these relations; the library defines (propositional-valued) $\leq$ as

$$x \leq y = \exists s{:}nat\; x + s = y.$$

It is hoped that the theorems given in the library are an adequate abstract interface for the relation $\leq$, i.e. any other lemmas are provable from the ones given without using the details of the definition. This can be enforced by using the command `Freeze Le`.

11.9.1 Propositional Inequality

The order relation $\leq$ is defined in one file, with its associated theorems. There is some redundancy e.g. `Le_resp_plus` is trivially implied by `Le_resp_plus_both`. Having both theorems available is useful in further development.

```
** Module lib_nat_Le Imports lib_nat_times_thms lib_nat_minus_thms
 Le = ... : nat->nat->Prop
 Le_plus_lemma1 = ... : {m,n|nat}Le m (plus m n)
 Le_plus_lemma2 = ... : {m,n|nat}Le n (plus m n)
 Le_suc_lemma = ... : {m,n|nat}(Le m (suc n))->(Le m n \/ Eq (suc n) m)
 Le_antisym = ... : {m,n:nat}(Le m n)->(Le n m)->Eq n m
 Le_suc = ... : {x:nat}Le x (suc x)
 Le_resp_suc = ... : {x,y:nat}(Le x y)->Le (suc x) (suc y)
 Le_resp_pred = ... : {x,y:nat}(Le (suc x) (suc y))->Le x y
```

```
Le_zero_n = ... : {n:nat}Le zero n
Le_incl_pred = ... : {x:nat}Le (pred x) x
not_Le_zero_n = ... : {n:nat}not (Le (suc n) zero)
Le_suc_pred = ... : {a,b|nat}(Le (suc a) b)->Le a (pred b)
eq_imp_Le = ... : {x,y:nat}(Eq x y)->Le x y
Le_trans = ... : {x,y,z|nat}(Le x y)->(Le y z)->Le x z
Le_resp_plus = ... : {x,y,z:nat}(Le x y)->Le (plus z x) (plus z y)
Le_resp_sub = ... : {x,y,z:nat}(Le (plus z x) (plus z y))->Le x y
Le_resp_plus_both = ... :
  {x,y,a,b:nat}(Le x y)->(Le a b)->Le (plus x a) (plus y b)
Le_resp_times = ... : {x,y,a:nat}(Le x y)->Le (times a x) (times a y)
Le_times_cancel = ... :
  {x,y,a:nat}(Le (times x (suc a)) (times y (suc a)))->Le x y
Le_resp_times_both = ... :
  {a,b,x,y:nat}(Le a b)->(Le x y)->Le (times a x) (times b y)
plus_inv_minus = ... : {m,n:nat}(Le m n)->Eq n (plus (minus n m) m)
```

11.9.2 Propositional Strict Inequality

Strict inequality is defined in the next file. A very similar collection of theorems is given to those above, with the addition of a couple of extra induction principles. These are in this file mainly because at this point all the necessary lemmas are available.

```
** Module lib_nat_Lt Imports lib_nat_minus_thms
 Lt = ... : nat->nat->Prop
 zero_Lt_suc_n = ... : {n:nat}Lt zero (suc n)
 n_Lt_suc_n = ... : {n:nat}Lt n (suc n)
 Lt_trans = ... : {l,m,n|nat}(Lt l m)->(Lt m n)->Lt l n
 not_n_Lt_zero = ... : {n:nat}not (Lt n zero)
 not_refl_Lt = ... : {x:nat}not (Lt x x)
 Lt_not_Eq = ... : {x,y:nat}(Lt x y)->not (Eq x y)
 Lt_zero = ... : {n:nat}(not (Eq n zero))->Lt zero n
 Lt_resp_suc = ... : {m,n:nat}(Lt m n)->Lt (suc m) (suc n)
 Lt_resp_pred = ... : {x,y:nat}(Lt (suc x) (suc y))->Lt x y
 Lt_pred = ... : {a,b|nat}(Lt a b)->Lt (pred b) b
 Lt_suc_pred = ... : {a,b|nat}(Lt (suc a) b)->Lt a (pred b)
 Lt_not_zero = ... : {a,b|nat}(Lt a b)->not (Eq b zero)
 Lt_pred_suc = ... : {a,b|nat}(Lt a (pred b))->Eq b (suc (pred b))
 Lt_pred_suc' = ... : {a,b|nat}(Lt (pred a) b)->Lt a (suc b)
 Lt_pred_suc'' = ... : {a,b|nat}(Lt a (pred b))->Lt (suc a) b
 n_Lt_suc_m_character = ... :
   {n,m|nat}(Lt n (suc m))->(Lt n m \/ Eq n m)
 plus_resp_Lt_l = ... :
```

```
  {l,m|nat}(Lt l m)->{n:nat}Lt (plus l n) (plus m n)
Lt_resp_plus = ... : {x,y,z:nat}(Lt x y)->Lt (plus z x) (plus z y)
Lt_resp_plus_both = ... :
  {x,y,a,b:nat}(Lt x y)->(Lt a b)->Lt (plus x a) (plus y b)
Lt_resp_plus_right = ... : {a,x,y|nat}(Lt x y)->Lt x (plus a y)
Lt_resp_times = ... :
  {x,y,a:nat}(Lt x y)->Lt (times (suc a) x) (times (suc a) y)
Lt_resp_sub = ... : {x,y,z:nat}(Lt (plus z x) (plus z y))->Lt x y
Lt_times_cancel = ... :
  {x,y,a:nat}(Lt (times x (suc a)) (times y (suc a)))->Lt x y
ze_Lt_imp_Eq_suc = ... : {m:nat}(Lt zero m)->Ex ([n:nat]Eq m (suc n))
exists_suc = ... :
  {P:nat->Prop}{n:nat}
  iff (Ex ([t:nat](Lt t (suc n) /\ P t)))
   (Ex ([t:nat](Lt t n /\ P t)) \/ P n)
complete_induction = ... :
  {P:nat->Prop}({n:nat}({x:nat}(Lt x n)->P x)->P n)->{m:nat}P m
well_founded_induction = ... :
  {t|Type}{f:t->nat}{P:t->Prop}
  ({x:t}({y:t}(Lt (f y) (f x))->P y)->P x)->{z:t}P z
nested_complete_induction = ... :
  {P:nat->nat->Prop}
  ({n,m:nat}({x,y:nat}(Lt x n \/ (Eq x n /\ Lt y m))->P x y)->P n m)->
  {n,m:nat}P n m
```

11.9.3 Propositional Relations

This file incorporates both propositional inequalities and gives several additional theorems involving both inequalities.

```
** Module lib_nat_Prop_rels Imports lib_nat_Lt lib_nat_Le
 Lt_Le_trans = ... : {x,y,z|nat}(Lt x y)->(Le y z)->Lt x z
 Le_Lt_trans = ... : {x,y,z|nat}(Le x y)->(Lt y z)->Lt x z
 Lt_imp_Le = ... : {x,y:nat}(Lt x y)->Le x y
 Lt_Le_resp_plus = ... :
   {x,y,a,b:nat}(Lt x y)->(Le a b)->Lt (plus x a) (plus y b)
 Le_Lt_resp_plus = ... :
   {x,y,a,b:nat}(Le x y)->(Lt a b)->Lt (plus x a) (plus y b)
 decide_nat = ... : {x,y:nat}(Le x y \/ Lt y x)
 Le_is_Lt_or_Eq = ... : {x,y:nat}(Le x y)->(Lt x y \/ Eq x y)
 zero_or_pos = ... : {n:nat}(Eq n zero \/ Lt zero n)
 Lt_imp_Le_suc = ... : {x,y:nat}(Lt x (suc y))->Le x y
 Le_imp_Lt_suc = ... : {x,y:nat}(Le x y)->Lt x (suc y)
```

```
Lt_resp_times_both = ... :
  {x,y,a,b:nat}(Lt x y)->(Lt a b)->Lt (times a x) (times b y)
Lt_imp_Le_suc' = ... : {x,y:nat}(Lt x y)->Le (suc x) y
Le_imp_Lt_suc' = ... : {x,y:nat}(Le (suc x) y)->Lt x y
minus_well_le = ... :
  {m,n:nat}(Lt zero m)->(Le m n)->Lt (plus (minus n m) m) (plus n m)
not_Le_imp_Lt = ... : {x,y:nat}(not (Le x y))->Lt y x
not_Lt_imp_Le = ... : {x,y:nat}(not (Lt x y))->Le y x
Le_minus_zero = ... : {a,b:nat}(Le a b)->Eq (minus a b) zero
Le_resp_minus = ... : {a,b,c:nat}(Le a b)->Le (minus c b) (minus c a)
```

11.9.4 Boolean-valued Relations

The order relations on nat are decidable so another way to represent them is via boolean valued functions. The following file defines boolean-valued strict and non-strict inequality plus the boolean-valued equality. Most of the propositional order relation theorems have a boolean counterpart, the most useful of which are included in the library.

```
** Module lib_nat_bool_rels Imports lib_nat_suc_thms lib_bool_thms
 lt = ... : nat->nat->bool
 le = ... : nat->nat->bool
 nat_eq = ... : nat->nat->bool
 zero_lt_suc = ... : {n:nat}is_true (lt zero (suc n))
 zero_le_suc = ... : {n:nat}is_true (le zero (suc n))
 not_n_lt_zero = ... : {n:nat}is_false (lt n zero)
 lt_resp_suc = ... : {n,m:nat}Eq (lt n m) (lt (suc n) (suc m))
 le_resp_suc = ... : {n,m:nat}Eq (le n m) (le (suc n) (suc m))
 not_n_lt_n = ... : {n:nat}is_false (lt n n)
 n_lt_suc_n = ... : {n:nat}is_true (lt n (suc n))
 not_lt_suc_is_suc = ... : {m,n:nat}(is_false (lt m (suc n)))->is_suc m
 lt_weak_trans = ... :
   {m,n|nat}(is_true (lt m n))->is_true (lt m (suc n))
 lt_left_suc = ... :
   {n,m:nat}(is_true (lt (suc n) m))->is_true (lt n m)
 lt_suc_character = ... :
   {m,n:nat}(is_true (lt n (suc m)))->(Eq n m \/ is_true (lt n m))
 lt_character = ... :
   {n,m:nat}
   ((is_true (lt n m) /\ is_false (lt m n)) \/
    ((is_true (lt m n) /\ is_false (lt n m)) \/ Eq n m))
 le_character = ... :
   {n,m:nat}
   ((is_true (le n m) /\ is_false (le m n)) \/
```

```
   ((is_true (le m n) /\ is_false (le n m)) \/ Eq n m))
 le_refl = ... : {n:nat}is_true (le n n)
 le_zero_imp_zero = ... : {n:nat}iff (is_true (le n zero)) (Eq n zero)
 le_not_lt_imp_Eq = ... :
   {i,n:nat}(is_true (le i n) /\ is_false (lt i n))->Eq i n
 lt_imp_le_suc = ... :
   {x,y|nat}(is_true (lt x y))->is_true (le (suc x) y)
 nat_eq_character = ... : {m,n:nat}iff (is_true (nat_eq m n)) (Eq m n)
 nat_eq_character' = ... :
   {m,n:nat}iff (is_false (nat_eq m n)) (not (Eq m n))
 nat_eq_sym = ... : {m,n:nat}Eq (nat_eq m n) (nat_eq n m)
 nat_eq_refl = ... : {n:nat}is_true (nat_eq n n)
 odd = ... : nat->bool
 even = ... : nat->bool
```

11.9.5 All order relations

The next file is very short in itself but it loads all the other order relation files. The theorems in it relate the propositional or logical relations with their boolean counterparts.

```
** Module lib_nat_rels Imports lib_nat_Prop_rels lib_nat_bool_rels
 le_true_imp_Le = ... : {n,m:nat}(Eq (le n m) true)->Le n m
 le_false_imp_Lt = ... : {n,m:nat}(Eq (le n m) false)->Lt m n
 Le_imp_le_true = ... : {n,m:nat}(Le n m)->Eq (le n m) true
 Lt_imp_le_false = ... : {n,m:nat}(Lt m n)->Eq (le n m) false
 lt_true_imp_Lt = ... : {n,m:nat}(Eq (lt n m) true)->Lt n m
 lt_false_imp_Le = ... : {n,m:nat}(Eq (lt n m) false)->Le m n
 Lt_imp_lt_true = ... : {n,m:nat}(Lt n m)->Eq (lt n m) true
 Le_imp_lt_false = ... : {n,m:nat}(Le n m)->Eq (lt m n) false
```

11.10 The predicate Even

In this section we define the predicate "Even", and prove some theorems characterising it.

```
** Module lib_nat_Even Imports lib_nat
 Even = ... : nat->Prop
 ev1 = ... : Even zero
 ev2 = ... : {n:nat}(Even n)->Even (suc (suc n))
 Even_elim_nondep = ... :
   {T:nat->Prop}(T zero)->({n:nat}(T n)->T (suc (suc n)))->{n:nat}
   (Even n)->T n
 Even_inv = ... :
   {n:nat}(Even n)->
```

```
   (Eq n zero \/ Ex ([x:nat](Eq n (suc (suc x)) /\ Even x)))
 Even_inv' = ... : {n:nat}(Even (suc (suc n)))->Even n
```

11.11 Max and Min

The final file for natural numbers defines the functions max and min and proves various theorems about them. It requires most of the other nat library files. The functions min and max are defined explicitly using double iteration rather than by showing that the order relation on nat is complete.

```
** Module lib_max_min Imports lib_nat_Prop_rels
 max = ... : nat->nat->nat
 min = ... : nat->nat->nat
 max_Le_left = ... : {x,y:nat}Le x (max x y)
 max_Le_right = ... : {x,y:nat}Le y (max x y)
 max_is_least = ... : {x,y,z:nat}(Le x z)->(Le y z)->Le (max x y) z
 min_Le_left = ... : {x,y:nat}Le (min x y) y
 min_Le_right = ... : {x,y:nat}Le (min x y) x
 min_is_greatest = ... : {x,y,z:nat}(Le z x)->(Le z y)->Le z (min x y)
 max_sym = ... : {n,m:nat}Eq (max n m) (max m n)
 max_assoc = ... : {m,n,o:nat}Eq (max (max m n) o) (max m (max n o))
 max_suc_pred = ... :
   {m,n:nat}Eq (max m (suc n)) (suc (max (pred m) n))
 min_sym = ... : {n,m:nat}Eq (min n m) (min m n)
 Le_min_left = ... : {x,y:nat}(Le x y)->Eq (min x y) x
 Le_min_right = ... : {x,y:nat}(Le y x)->Eq (min x y) y
 Le_max_right = ... : {x,y:nat}(Le x y)->Eq (max x y) y
 Le_max_left = ... : {x,y:nat}(Le y x)->Eq (max x y) x
 min_idemp = ... : {a:nat}Eq (min a a) a
 max_resp_times = ... :
   {x,y,a:nat}Eq (max (times a x) (times a y)) (times a (max x y))
 min_resp_times = ... :
   {x,y,a:nat}Eq (min (times a x) (times a y)) (times a (min x y))
 max_resp_plus = ... :
   {a,b,x:nat}Eq (max (plus a x) (plus b x)) (plus (max a b) x)
 min_resp_plus = ... :
   {a,b,x:nat}Eq (min (plus a x) (plus b x)) (plus (min a b) x)
```

11.12 Division

```
** Module lib_nat_div Imports lib_nat_rels lib_bool_thms
 half = ... : nat->nat
 even_pred = ... :
```

```
    {n|nat}{b|bool}(Eq (even (suc n)) (inv b))->Eq (even n) b
  half_even = ... :
    {n:nat}
    ((is_true (even n))->Eq (times two (half n)) n /\
     (is_false (even n))->Eq (suc (times two (half n))) n)
```

11.13 Exponentiation

```
 ** Module lib_nat_exp_thms Imports lib_nat_times_thms
  exp_one = ... : {n:nat}Eq (exp n one) n
```

12 List

Another large section of the library is the directory `lib_list`. There are several files. The most basic is `lib_list.l` which contains just the definition of the inductive type list. The four files `lib_list_Adjoin`, `lib_list_Distin`, `lib_list_Member`, `lib_list_Perm` and `lib_list_PermII` define the predicates Adjoin, Distin, Member and two version of Perm respectively and prove theorems about them. The other files in this directory are divided according to their dependency on other datatypes. The file `lib_list_basics` contains functions and theorems which do not depend on any other datatypes, `lib_list_bool` depends on the boolean datatype. `lib_list_partial` defines some properly partial functions using the sum and empty datatypes to construct "partial" result types, the "length" file defines functions dependent on the natural number type, principally "length" but also a couple of other functions. `lib_list_misc` contains the one function and theorem which require both sum and natural numbers.

12.1 List Definitions

The basic file contains the definition of list and the usual elimination, reduction, iteration and double elimination rules plus the characterisation theorems.

```
 ** Module lib_list Imports lib_eq_basics
 $list : (Type)->Type(list)
 $nil : {t:Type}list t
 $cons1 : {t:Type}t->(list t)->list t
 $list_elim :
    {t:Type}{C_list:(list t)->Type}(C_list (nil t))->
    ({x1:t}{x2:list t}(C_list x2)->C_list (cons1 t x1 x2))->{z:list t}
    C_list z
[[t:Type][C_list:(list t)->Type][f_nil:C_list (nil t)]
 [f_cons1:{x1:t}{x2:list t}(C_list x2)->C_list (cons1 t x1 x2)][x1:t]
 [x2:list t]
```

```
   list_elim t C_list f_nil f_cons1 (nil t)  ==>  f_nil
|| list_elim t C_list f_nil f_cons1 (cons1 t x1 x2)  ==>
   f_cons1 x1 x2 (list_elim t C_list f_nil f_cons1 x2)]

 cons = ... : {t|Type}t->(list t)->list t
 list_rec = ... : {s|Type}{t|Type}t->(s->(list s)->t->t)->(list s)->t
 list_iter = ... : {s|Type}{t|Type}t->(s->t->t)->(list s)->t
 list_ind = ... :
   {s|Type}{P:(list s)->Prop}(P (nil s))->
   ({x:s}{l:list s}(P l)->P (cons x l))->{l:list s}P l
 list_double_elim = ... :
   {s|Type}{T:(list s)->(list s)->Type}(T (nil s) (nil s))->
   ({x:s}{l:list s}(T (nil s) l)->T (nil s) (cons x l))->
   ({x:s}{l:list s}({l':list s}T l l')->T (cons x l) (nil s))->
   ({x:s}{l:list s}({l':list s}T l l')->{y:s}{l':list s}
    (T (cons x l) l')->T (cons x l) (cons y l'))->{l,l':list s}T l l'
 is_nil = ... : {t|Type}(list t)->Prop
 is_cons = ... : {t|Type}(list t)->Prop
 nil_or_cons = ... : {t|Type}{l:list t}(is_nil l \/ is_cons l)
 nil_not_cons = ... :
   {t|Type}{l:list t}{x:t}not (Eq (nil t) (cons x l))
```

12.2 List Basics

This file defines some basic list functions—where basic really just means pure list functions which do not require any other datatype. These functions include a head function—`hd` which takes an extra argument which is the value returned for a empty list, tail, fold, append, map and reverse. Some theorems about these functions are also given (but not enough).

```
** Module lib_list_basics Imports lib_list
 singleton = ... : {t|Type}t->list t
 hd = ... : {t|Type}t->(list t)->t
 tail = ... : {t|Type}(list t)->list t
 fold = ... : {s|Type}{t|Type}(t->s->t)->t->(list s)->t
 foldright = ... : {B|Type}{A|Type}(A->B->B)->(list A)->B->B
 foldleft = ... : {B|Type}{A|Type}(A->B->B)->(list A)->B->B
 map = ... : {s|Type}{t|Type}(s->t)->(list s)->list t
 append = ... : {t|Type}(list t)->(list t)->list t
 tack = ... : {t|Type}t->(list t)->list t
 reverse = ... : {t|Type}(list t)->list t
 flatten = ... : {t|Type}(list (list t))->list t
 cons_injective = ... :
```

```
    {t|Type}{k,l|list t}{x,y|t}(Eq (cons x k) (cons y l))->
    Eq (tail (cons x k)) (tail (cons y l))
  cons_inj = ... :
    {s|Type}{a,b|s}{xs,ys|list s}(Eq (cons a xs) (cons b ys))->
    (Eq a b /\ Eq xs ys)
  l_not_cons_l = ... : {t|Type}{l:list t}{x:t}not (Eq l (cons x l))
  append_cons = ... :
    {t|Type}{k,l:list t}{x:t}
    Ex2 ([y:t][j:list t]Eq (append k (cons x l)) (cons y j))
  nil_not_append_cons = ... :
    {t|Type}{x:t}{k,l:list t}not (Eq (nil t) (append k (cons x l)))
  append_assoc = ... :
    {t|Type}{j,k,l:list t}
    Eq (append j (append k l)) (append (append j k) l)
  append_nil = ... : {t|Type}{l:list t}Eq (append l (nil t)) l
  append_reverse = ... :
    {t|Type}{l,m:list t}
    Eq (reverse (append l m)) (append (reverse m) (reverse l))
  append_strict = ... :
    {t|Type}{k,l:list t}(Eq (append k l) (nil t))->
    (Eq k (nil t) /\ Eq l (nil t))
  double_reverse_is_identity = ... :
    {t|Type}{l:list t}Eq (reverse (reverse l)) l
  foldright_append_lem = ... :
    {s|Type}{t|Type}{g:s->t->t}{G,H:list s}{b:t}
    Eq (foldright g (append G H) b) (foldright g G (foldright g H b))
  foldleft_append_lem = ... :
    {s|Type}{t|Type}{g:s->t->t}{G,H:list s}{b:t}
    Eq (foldleft g (append G H) b) (foldleft g H (foldleft g G b))
```

12.3 List functions with booleans

This file contains list functions and theorems involving the booleans. These include the boolean valued test for nil, test for membership give a boolean-valued equality over the type of elements of the list, list subtraction, universal and existential extensions of a predicate to a list of values.

```
 ** Module lib_list_bool Imports lib_list_basics lib_bool_thms
  nil_test = ... : {s|Type}(list s)->bool
  exist_list = ... : {t|Type}(t->bool)->(list t)->bool
  all_list = ... : {t|Type}(t->bool)->(list t)->bool
  member = ... : {t|Type}(t->t->bool)->t->(list t)->bool
  list_eq = ... : {t|Type}(t->t->bool)->(list t)->(list t)->bool
```

```
del_first = ... : {t|Type}(t->t->bool)->t->(list t)->list t
del_all = ... : {t|Type}(t->t->bool)->t->(list t)->list t
list_sub = ... : {A|Type}(Type)->(A->bool)->A->(list A)->list A
exist_list_append_lem = ... :
  {t|Type}{P:t->bool}{j,k:list t}
  Eq (exist_list P (append j k))
   (orelse (exist_list P j) (exist_list P k))
member_append_lem = ... :
  {t|Type}{eq_on_t:t->t->bool}{x:t}{j,k:list t}
  Eq (member eq_on_t x (append j k))
   (orelse (member eq_on_t x j) (member eq_on_t x k))
nil_test_true_equiv_Eq_nil = ... :
  {t|Type}{l:list t}iff (Eq (nil_test l) true) (Eq l (nil t))
nil_test_false_equiv_Eq_cons = ... :
  {t|Type}{l:list t}
  iff (Eq (nil_test l) false)
   (Ex ([a:t]Ex ([m:list t]Eq l (cons a m))))
nil_test_on_append = ... :
  {t|Type}{l,m:list t}{a:t}Eq (nil_test (append l (cons a m))) false
```

12.4 Partial List functions

This file contains partial list functions whose type is lifted using the sum and empty datatype, e.g. head has type $list\alpha \rightarrow \alpha+$. Other functions defined here include "last", which gives the last element of a list and "prune" which deletes the last element of a list. Various theorems about these functions are proved.

```
** Module lib_list_partial Imports lib_list_bool lib_unit lib_sum
 head = ... : {t|Type}(list t)->sum unit t
 last = ... : {s|Type}(list s)->sum unit s
 prune = ... : {s|Type}(list s)->sum unit (list s)
 last_resp_cons = ... :
   {t|Type}{l:list t}{a,b:t}
   Eq (last (cons a l)) (last (cons b (cons a l)))
 last_resp_append = ... :
   {t|Type}{l,m:list t}{a:t}
   Eq (last (append l (cons a m))) (last (cons a m))
 last_gives_last = ... :
   {t|Type}{l:list t}{a:t}Eq (last (append l (cons a (nil t)))) (in2 a)
 prune_resp_cons = ... :
   {t|Type}{l:list t}{a,b:t}
   Eq (prune (cons b (cons a l)))
    (case ([_:unit]in1 void) ([l'5:list t]in2 (cons b l'5))
```

```
      (prune (cons a l)))
  prune_non_empty_is_list = ... :
    {t|Type}{l:list t}{a:t}Ex ([m:list t]Eq (prune (cons a l)) (in2 m))
  prune_resp_append = ... :
    {t|Type}{l,m:list t}{a:t}
    Eq (prune (append l (cons a m)))
     (case ([_:unit]in1 void) ([x:list t]in2 (append l x))
       (prune (cons a m)))
  prune_deletes_tip = ... :
    {t|Type}{x:t}{l:list t}
    Eq (prune (append l (cons x (nil t)))) (in2 l)
```

12.5 Length-like list functions

This section contains the list functions which require the natural number datatype; the most important of these functions is the "length" function which returns the length of a list. Also given is the nth function which returns the element of a list and a function called `build_list` which constructs the list corresponding to the first "n" elements of a stream represented as a function from nat to the elements of some set.

```
 ** Module lib_list_length Imports lib_list_basics lib_nat_Lt
  length = ... : {s|Type}(list s)->nat
  nth = ... : nat->{t|Type}(list t)->t->t
  build_list = ... : {s|Type}(nat->s)->nat->list s
  list_length_induction = ... :
    {t|Type}{P:(list t)->Prop}
    ({l:list t}({x:list t}(Lt (length x) (length l))->P x)->P l)->
    {k:list t}P k
  length_zero_nil = ... :
    {s|Type}{l:list s}(Eq (length l) zero)->Eq l (nil s)
  length_map = ... :
    {s|Type}{t|Type}{l:list s}{f:s->t}Eq (length (map f l)) (length l)
  length_build_list = ... :
    {s|Type}{f:nat->s}{n:nat}Eq (length (build_list f n)) n
  length_append = ... :
    {s|Type}{l1,l2:list s}
    Eq (length (append l1 l2)) (plus (length l1) (length l2))
```

12.6 Other list functions

This section (so far) contains the boolean-valued function which tests to see of two lists have the same length. This function requires both the boolean and the natural number datatypes and is separated to reduce the dependencies.

```
** Module lib_list_misc Imports lib_list_length lib_list_partial
 same_length = ... : {s|Type}{t|Type}(list s)->(list t)->bool
 Eq_length_imp_same_length_true = ... :
   {t|Type}{l,m:list t}(Eq (length l) (length m))->
   is_true (same_length l m)
```

12.7 Lists and products

In this file the functions and theorems involving cross-product and list are collected together. The function `prod_list` creates a list of all possible pairs from the two original lists.

```
** Module lib_list_prod Imports lib_list_length lib_list_bool lib_prod
 assoc = ... : {A|Type}{B|Type}(A->bool)->B->(list (prod A B))->B
 lib_list_dom = ... : {A|Type}{B|Type}(list (prod A B))->list A
 cod = ... : {A|Type}{B|Type}(list (prod A B))->list B
 zip = ... : {A|Type}{B|Type}(list A)->(list B)->list (prod A B)
 prod_list = ... : {s|Type}{t|Type}(list s)->(list t)->list (prod s t)
 length_prod = ... :
   {s|Type}{t|Type}{l1:list s}{l2:list t}
   Eq (length (prod_list l1 l2)) (times (length l1) (length l2))
```

12.8 The Relation Adjoin

Intuitively, `Adjoin a k l` holds if it is possible to insert the element `a` in the list `k` and obtain the list `l`. This relation can be used to define permutations, as seen below.

```
 ** Module lib_list_Adjoin Imports lib_list_length
 $Adjoin : {s|Type}s->(list s)->(list s)->Prop
 $ad1 : {s|Type}{a:s}{xs:list s}Adjoin a xs (cons a xs)
 $ad2 :
    {s|Type}{xs|list s}{a|s}{axs|list s}(Adjoin a xs axs)->{b:s}
    Adjoin a (cons b xs) (cons b axs)
 $Adjoin_elim :
    {s|Type}{C_Adjoin:s->(list s)->(list s)->Prop}
    ({a:s}{xs:list s}C_Adjoin a xs (cons a xs))->
    ({xs|list s}{a|s}{axs|list s}(Adjoin a xs axs)->{b:s}
     (C_Adjoin a xs axs)->C_Adjoin a (cons b xs) (cons b axs))->{x1|s}
    {x2,x3|list s}(Adjoin x1 x2 x3)->C_Adjoin x1 x2 x3
[[s|Type][C_Adjoin:s->(list s)->(list s)->Prop]
 [f_ad1:{a1:s}{xs1:list s}C_Adjoin a1 xs1 (cons a1 xs1)]
 [f_ad2:{xs|list s}{a|s}{axs|list s}(Adjoin a xs axs)->{b:s}
  (C_Adjoin a xs axs)->C_Adjoin a (cons b xs) (cons b axs)][a1:s]
```

```
[xs1:list s][xs|list s][a|s][axs|list s][x1:Adjoin a xs axs][b:s]
   Adjoin_elim C_Adjoin f_ad1 f_ad2 (ad1 a1 xs1)  ==>  f_ad1 a1 xs1
|| Adjoin_elim C_Adjoin f_ad1 f_ad2 (ad2 x1 b)  ==>
   f_ad2 x1 b (Adjoin_elim C_Adjoin f_ad1 f_ad2 x1)]

 Adjoin_inv = ... :
   {s|Type}{a|s}{xs,ys|list s}(Adjoin a xs ys)->
   (Eq ys (cons a xs) \/
    Ex ([xs':list s]
        Ex ([ys':list s]
            Ex ([c:s]
                (Eq xs (cons c xs') /\ Eq ys (cons c ys') /\
                 Adjoin a xs' ys')))))
 Adjoin_inv_corr1 = ... :
   {s|Type}{a,b|s}{xs,ys|list s}(Adjoin a (cons b xs) ys)->
   (Eq ys (cons a (cons b xs)) \/
    Ex ([ys':list s](Eq ys (cons b ys') /\ Adjoin a xs ys')))
 Adjoin_inv_corr2 = ... :
   {s|Type}{a,b|s}{xs,ys|list s}(Adjoin a xs (cons b ys))->
   ((Eq a b /\ Eq xs ys) \/
    Ex ([xs':list s](Eq xs (cons b xs') /\ Adjoin a xs' ys)))
 Adjoin_inv_corr3 = ... :
   {s|Type}{a|s}{xs|list s}(Adjoin a (nil s) xs)->
   Eq xs (cons a (nil s))
 Adjoin_inv_corr4 = ... :
   {s|Type}{a|s}{xs|list s}not (Adjoin a xs (nil s))
 Adjoin_inv_corr5 = ... :
   {s|Type}{a|s}{xs|list s}(Adjoin a xs (cons a (nil s)))->
   Eq xs (nil s)
 Adjoin_com1 = ... :
   {s|Type}{a,b|s}{xs,us,ys|list s}(Adjoin a xs ys)->(Adjoin b ys us)->
   Ex ([zs:list s](Adjoin b xs zs /\ Adjoin a zs us))
 Adjoin_com2 = ... :
   {s|Type}{a,b|s}{xs,us,zs|list s}(Adjoin b zs xs)->(Adjoin a zs us)->
   Ex ([ys:list s](Adjoin a xs ys /\ Adjoin b us ys))
 Adjoin_com3 = ... :
   {s|Type}{a,b|s}{xs,us,ys|list s}(Adjoin a xs ys)->(Adjoin b us ys)->
   ((Eq us xs /\ Eq b a) \/
    Ex ([zs:list s](Adjoin b zs xs /\ Adjoin a zs us)))
 Adjoin_length_suc = ... :
   {s|Type}{a|s}{l,al|list s}(Adjoin a l al)->
   Eq (length al) (suc (length l))
```

12.9 Membership

Using the relation `Adjoin` we define the membership relation between elements of some type and lists over that type.

```
** Module lib_list_Member Imports lib_list_Adjoin lib_list_prod
 Member = ... : {s|Type}s->(list s)->Prop
 not_Member_nil = ... : {s|Type}{a|s}not (Member a (nil s))
 Mem_ind = ... :
   {s|Type}{l|list s}{a,b|s}
   iff (Member b (cons a l)) (Member b l \/ Eq a b)
 Member_Adjoin = ... :
   {s|Type}{l,al|list s}{a,b|s}(not (Eq a b))->(Adjoin a l al)->
   (Member b al)->Member b l
 Member_map = ... :
   {s|Type}{u|Type}{l:list s}{a:s}{f:s->u}(Member a l)->
   Member (f a) (map f l)
 Member_map_inv = ... :
   {s|Type}{u|Type}{l:list s}{b:u}{f:s->u}(Member b (map f l))->
   Ex ([a:s](Member a l /\ Eq (f a) b))
 Member_append = ... :
   {s|Type}{l1,l2:list s}{a:s}
   iff (Member a (append l1 l2)) (Member a l1 \/ Member a l2)
 Member_prod = ... :
   {s|Type}{u|Type}{l1:list s}{l2:list u}{p:prod s u}
   iff (Member p (prod_list l1 l2))
    (Member (Fst p) l1 /\ Member (Snd p) l2)
 Member_build = ... :
   {s|Type}{f:nat->s}{n:nat}{a:s}
   iff (Member a (build_list f n)) (Ex ([k:nat](Lt k n /\ Eq (f k) a)))
```

12.10 Distinctness

We define the property `Distin`, that all the elements of a list are distinct.

```
** Module lib_list_Distin Imports lib_list_Member lib_nat_Le
$Distin : {s|Type}(list s)->Prop
$Dis1 : {s|Type}Distin (nil s)
$Dis2 :
   {s|Type}{a:s}{xs:list s}(not (Member a xs))->(Distin xs)->
   Distin (cons a xs)
$Distin_elim :
   {s|Type}{C_Distin:(list s)->Prop}(C_Distin (nil s))->
   ({a:s}{xs:list s}(not (Member a xs))->(Distin xs)->(C_Distin xs)->
```

```
     C_Distin (cons a xs))->{x1|list s}(Distin x1)->C_Distin x1
[[s|Type][C_Distin:(list s)->Prop][f_Dis1:C_Distin (nil s)]
 [f_Dis2:{a:s}{xs:list s}(not (Member a xs))->(Distin xs)->
  (C_Distin xs)->C_Distin (cons a xs)][a:s][xs:list s]
 [x1:not (Member a xs)][x2:Distin xs]
    Distin_elim C_Distin f_Dis1 f_Dis2 (Dis1|s)  ==>  f_Dis1
 || Distin_elim C_Distin f_Dis1 f_Dis2 (Dis2 a xs x1 x2)  ==>
    f_Dis2 a xs x1 x2 (Distin_elim C_Distin f_Dis1 f_Dis2 x2)]

  Distin_inv = ... :
    {s|Type}{a|s}{l|list s}(Distin (cons a l))->
    (not (Member a l) /\ Distin l)
  Distin_append = ... :
    {s|Type}{l1,l2:list s}(Distin l1)->(Distin l2)->
    ({a:s}(Member a l1)->not (Member a l2))->Distin (append l1 l2)
  Distin_inj_map = ... :
    {s|Type}{u|Type}{l:list s}{f:s->u}
    ({a1,a2:s}(Member a1 l)->(Member a2 l)->(Eq (f a1) (f a2))->
     Eq a1 a2)->(Distin l)->Distin (map f l)
  Distin_prod = ... :
    {s|Type}{u|Type}{l1:list s}{l2:list u}(Distin l1)->(Distin l2)->
    Distin (prod_list l1 l2)
  Distin_inj_build = ... :
    {s|Type}{f:nat->s}{n:nat}
    ({k1,k2:nat}(Lt k1 n)->(Lt k2 n)->(Eq (f k1) (f k2))->Eq k1 k2)->
    Distin (build_list f n)
  Distin_sublist_Le_length = ... :
    {s|Type}{xs,ys:list s}(Distin xs)->
    ({a:s}(Member a xs)->Member a ys)->Le (length xs) (length ys)
```

12.11 Permutation

We define the relation Perm that one list is a permutation of another.

```
** Module lib_list_Perm Imports lib_list_Distin
$Perm : {s|Type}(list s)->(list s)->Prop
$perm1 : {s|Type}Perm (nil s) (nil s)
$perm2 :
    {s|Type}{xs,ys:list s}(Perm xs ys)->{a:s}{axs:list s}
    (Adjoin a xs axs)->{ays:list s}(Adjoin a ys ays)->Perm axs ays
$Perm_elim :
    {s|Type}{C_Perm:(list s)->(list s)->Prop}(C_Perm (nil s) (nil s))->
    ({xs,ys:list s}(Perm xs ys)->{a:s}{axs:list s}(Adjoin a xs axs)->
```

```
     {ays:list s}(Adjoin a ys ays)->(C_Perm xs ys)->C_Perm axs ays)->
    {x1,x2|list s}(Perm x1 x2)->C_Perm x1 x2
[[s|Type][C_Perm:(list s)->(list s)->Prop]
 [f_perm1:C_Perm (nil s) (nil s)]
 [f_perm2:{xs,ys:list s}(Perm xs ys)->{a:s}{axs:list s}
  (Adjoin a xs axs)->{ays:list s}(Adjoin a ys ays)->(C_Perm xs ys)->
  C_Perm axs ays][xs,ys:list s][x1:Perm xs ys][a:s][axs:list s]
 [x2:Adjoin a xs axs][ays:list s][x3:Adjoin a ys ays]
    Perm_elim C_Perm f_perm1 f_perm2 (perm1|s)  ==>  f_perm1
 || Perm_elim C_Perm f_perm1 f_perm2 (perm2 xs ys x1 a axs x2 ays x3)
    ==>
    f_perm2 xs ys x1 a axs x2 ays x3
     (Perm_elim C_Perm f_perm1 f_perm2 x1)]

  Perm_fact = ... :
    {s|Type}{xs,ys|list s}{a|s}(Perm xs ys)->
    Perm (cons a xs) (cons a ys)
  Perm_refl = ... : {s|Type}{xs:list s}Perm xs xs
  Perm_sym = ... : {s|Type}{xs,ys|list s}(Perm xs ys)->Perm ys xs
  Perm_Adjoin_com1 = ... :
    {s|Type}{a|s}{vs,xs,ys,zs|list s}(Perm vs xs)->(Adjoin a xs ys)->
    (Adjoin a zs ys)->Perm vs zs
  Perm_Adjoin_com2 = ... :
    {s|Type}{a|s}{xs,ys,zs,us|list s}(Adjoin a xs ys)->(Perm ys zs)->
    (Adjoin a us zs)->Perm xs us
  Perm_Adjoin_com2_corr = ... :
    {s|Type}{a|s}{xs,ys|list s}(Perm (cons a xs) (cons a ys))->
    Perm xs ys
  Perm_Adjoin_com3 = ... :
    {s|Type}{a|s}{xs,zs,ys|list s}(Adjoin a xs ys)->(Perm ys zs)->
    Ex ([us:list s](Perm xs us /\ Adjoin a us zs))
  Perm_trans = ... :
    {s|Type}{xs,ys,zs|list s}(Perm xs ys)->(Perm ys zs)->Perm xs zs
  Perm_of_Distin = ... :
    {s|Type}{xs,ys:list s}(Distin xs)->
    ({a:s}(Member a xs)->Member a ys)->(Eq (length xs) (length ys))->
    Perm xs ys
  surjective_imp_Perm = ... :
    {s|Type}{l:list s}{f,g:s->s}(Distin l)->
    ({x:s}(Member x l)->Member (g x) l)->({x:s}Eq (f (g x)) x)->
    Perm l (map f l)
```

There is also an alternative, much more extensive but incompatible implementation of permutations.

```
** Module lib_list_PermII Imports lib_list_length lib_rel_closure
 insert = ... : {A|Type(0)}A->(list A)->(list A)->list A
 insertnil = ... :
   {A|Type(0)}{a|A}{h,k|list A}(Eq (insert a h k) (nil A))->absurd
 insertsingleton = ... :
   {A|Type(0)}{a,b|A}{h,k|list A}(Eq (insert a h k) (cons b (nil A)))->
   and3 (Eq (nil A) h) (Eq a b) (Eq (nil A) k)
 insertapp = ... :
   {A|Type(0)}{a|A}{h,k|list A}{l,m:list A}
   (Eq (insert a h k) (append l m))->
   (({phi:Prop}
    ({n:list A}(Eq (insert a h n) l)->(Eq (append n m) k)->phi)->phi)
    \/ {phi:Prop}
    ({n:list A}(Eq (append l n) h)->(Eq (insert a n k) m)->phi)->phi)
 inseqins = ... :
   {A|Type(0)}{a|A}{h,k,l,m|list A}(Eq (insert a h k) (insert a l m))->
   (({phi:Prop}
    ({n:list A}(Eq (insert a h n) l)->(Eq (insert a n m) k)->phi)->phi)
    \/
    ((Eq l h /\ Eq k m) \/ {phi:Prop}
     ({n:list A}(Eq (insert a l n) h)->(Eq (insert a n k) m)->phi)->
     phi))
 swap = ... : {A|Type(0)}(Rel (list A) (list A))->Prop
 conscl = ... :
   {A|Type(0)}A->(Rel (list A) (list A))->(list A)->(list A)->Prop
 consClosed = ... : {A|Type(0)}(Rel (list A) (list A))->Prop
 iclSwap = ... : {A|Type(0)}IntersectionClosed (swap|A)
 iclConscl = ... : {A|Type(0)}IntersectionClosed (consClosed|A)
 Fbase = ... : {A|Type(0)}Pred (Rel (list A) (list A))
 iclBase = ... : {A|Type(0)}IntersectionClosed (Fbase|A)
 PermBase = ... : {A|Type(0)}Rel (list A) (list A)
 swapBase = ... : {A|Type(0)}swap (PermBase|A)
 consclBase = ... : {A|Type(0)}consClosed (PermBase|A)
 Perm = ... : {A|Type(0)}Rel (list A) (list A)
 equivPerm = ... : {A|Type(0)}equiv (Perm|A)
 reflPerm = ... : {A|Type(0)}refl (Perm|A)
 symPerm = ... : {A|Type(0)}sym (Perm|A)
 transPerm = ... : {A|Type(0)}trans (Perm|A)
 reflPerm' = ... : {A|Type(0)}SubRel (Eq|(list A)) (Perm|A)
 swapPerm = ... : {A|Type(0)}swap (Perm|A)
```

```
consclPerm = ... : {A|Type(0)}consClosed (Perm|A)
appcl = ... :
  {A|Type(0)}(list A)->(Rel (list A) (list A))->(list A)->(list A)->
  Prop
appClosed = ... : {A|Type(0)}(Rel (list A) (list A))->Prop
iclAppcl = ... : {A|Type(0)}IntersectionClosed (appClosed|A)
appclPerm = ... : {A|Type(0)}appClosed (Perm|A)
recPerm = ... :
  {A|Type(0)}{l,m:list A}(Perm l m)->{R:Rel (list A) (list A)}
  ({l'6:list A}R l'6 l'6)->
  ({l'7,m'8:list A}(Perm l'7 m'8)->(R l'7 m'8)->R m'8 l'7)->
  ({l'8,m'9,n:list A}(Perm l'8 m'9)->(R l'8 m'9)->(Perm m'9 n)->
   (R m'9 n)->R l'8 n)->
  ({l'9,m'10:list A}R (append l'9 m'10) (append m'10 l'9))->
  ({a:A}{m'11,n:list A}(Perm m'11 n)->(R m'11 n)->
   R (cons a m'11) (cons a n))->R l m
PermE = ... :
  {A|Type(0)}{R:Rel (list A) (list A)}({l,m|list A}(Eq l m)->R l m)->
  ({l,m|list A}(Perm l m)->(R l m)->R m l)->
  ({l,m,n|list A}(Perm l m)->(R l m)->(Perm m n)->(R m n)->R l n)->
  ({l,m|list A}R (append l m) (append m l))->
  ({a|A}{m,n|list A}(Perm m n)->(R m n)->R (cons a m) (cons a n))->
  SubRel (Perm|A) R
indPerm = ... : {A|Type(0)}Rel (list A) (list A)
PermIsInductive = ... : {A|Type(0)}SubRel (Perm|A) (indPerm|A)
PermRrespR = ... :
  {A|Type(0)}{B|Type}{R:Rel B (list A)}
  ({l,m:list A}{h:B}(R h (append l m))->R h (append m l))->
  ({b:A}{l,m:list A}(Perm l m)->({h:B}(R h l)->R h m)->{k:B}
   (R k (cons b l))->R k (cons b m))->SubRel (Perm|A) (impliesRel R R)
PermLrespR = ... :
  {A|Type(0)}{B|Type}{R:Rel (list A) B}
  ({l,m:list A}{h:B}(R (append l m) h)->R (append m l) h)->
  ({b:A}{l,m:list A}(Perm l m)->({h:B}(R l h)->R m h)->{k:B}
   (R (cons b l) k)->R (cons b m) k)->
  SubRel (Perm|A) (coimpliesRel R R)
PermPredE = ... :
  {A|Type(0)}{P:Pred (list A)}
  ({l,m:list A}(P (append l m))->P (append m l))->
  ({b:A}{l,m:list A}(Perm l m)->(iff (P l) (P m))->(P (cons b l))->
   P (cons b m))->{l,m:list A}(Perm l m)->(P l)->P m
nilPermlemma = ... :
  {A|Type(0)}{l,m|list A}(Perm l m)->(Eq (nil A) l)->Eq (nil A) m
```

```
nilPerm = ... : {A|Type(0)}{l:list A}(Perm (nil A) l)->Eq (nil A) l
singletonPermlemma = ... :
  {A|Type(0)}{l,m:list A}(Perm l m)->{a:A}(Eq (singleton a) l)->
  Ex ([b:A](Eq a b /\ Eq (singleton b) m))
singletonPerm = ... :
  {A|Type(0)}{a,b:A}(Perm (singleton a) (singleton b))->Eq a b
PermRespLength = ... :
  {A|Type(0)}{l,m|list A}(Perm l m)->Eq (length l) (length m)
PermResidue = ... : {A|Type(0)}A->Rel (list A) (list A)
insResidue = ... :
  {A|Type(0)}{a:A}{h,k,l:list A}(Eq (insert a h k) l)->
  PermResidue a (append h k) l
swapResidue = ... :
  {A|Type(0)}{a:A}{l,m,n:list A}(PermResidue a n (append l m))->
  PermResidue a n (append m l)
consclResidue = ... :
  {A|Type(0)}{a,b:A}{l,m:list A}(Perm l m)->
  ({h:list A}(PermResidue a h l)->PermResidue a h m)->{k:list A}
  (PermResidue a k (cons b l))->PermResidue a k (cons b m)
funopResidue = ... :
  {A|Type(0)}{a:A}{h,k,l:list A}(PermResidue a h l)->
  (PermResidue a k l)->Perm h k
transResiduelemma = ... :
  {A|Type(0)}{a:A}{l,m,h|list A}(Perm l m)->(Eq (cons a h) l)->
  PermResidue a h m
heredPermlemma = ... :
  {A|Type(0)}{a:A}{l,m|list A}(Perm (cons a l) (cons a m))->Perm l m
hereditary = ... : {A|Type(0)}(Rel (list A) (list A))->Prop
heredPerm = ... : {A|Type(0)}hereditary (Perm|A)
transposePerm = ... :
  {A|Type(0)}{a,b|A}{l,m,n|list A}
  Perm (insert a l (insert b m n)) (insert b l (insert a m n))
```

12.12 Sorted Lists

This file sets up the predicate for sorted-ness of a list inductively using `list_elim`, from the predicate `lelist`, saying that a given element is less than some other list. The main purpose of the file is to establish a derived induction principle for sorted lists, `sorted_list_induction`, which was used to give a proof of insert sort in McKinna's (1992) PhD thesis.

```
** Module lib_list_sorted Imports lib_list_PermII
 lengthisahomomorphismcons = ... :
```

```
  {A|Type}{l|list A}{n|nat}(Eq (suc n) (length l))->
  Ex ([a:A]Eq l (cons a (tail l)))
Rlist = ... : {A|Type}{B|Type}(Rel A B)->A->(list B)->Prop
Sorted = ... : {A|Type}(Rel A A)->Pred (list A)
SortSpec = ... : {A|Type}(Rel A A)->Rel (list A) (list A)
sortedlistinduction = ... :
  {A|Type}{R:Rel A A}{B|Type}{phi:Rel (list A) B}{n|B}
  {c|A->(list A)->B->B}(phi (nil A) n)->
  ({a|A}{l|list A}{b|B}(Sorted R (cons a l))->(phi l b)->
   phi (cons a l) (c a l b))->{l:list A}(Sorted R l)->
  phi l (list_rec n ([a:A][l'12:list A][b:B]c a l'12 b) l)
nilSorted = ... : {A|Type}{R:Rel A A}Sorted R (nil A)
nilRlist = ... : {A|Type}{R:Rel A A}{c|A}Rlist R c (nil A)
heredSortedlemma = ... :
  {A|Type}{R:Rel A A}{b|A}{n|list A}(Sorted R (cons b n))->Sorted R n
SortedImpliesRlist = ... :
  {A|Type}{R:Rel A A}{b|A}{n|list A}(Sorted R (cons b n))->Rlist R b n
heredRlistlemma = ... :
  {A|Type}{R:Rel A A}{b,c|A}{n|list A}(Rlist R c (cons b n))->
  Rlist R c n
heredRlist1 = ... :
  {A|Type}{R:Rel A A}{c|A}{m,n|list A}(Rlist R c (append m n))->
  Rlist R c n
heredRlist2 = ... :
  {A|Type}{R:Rel A A}{c|A}{m,n|list A}(Rlist R c (append m n))->
  Rlist R c m
appclRlist = ... :
  {A|Type}{R:Rel A A}{c|A}{m,n|list A}(Rlist R c m)->(Rlist R c n)->
  Rlist R c (append m n)
heredSorted = ... :
  {A|Type}{R:Rel A A}{m,n|list A}(Sorted R (append m n))->
  (Sorted R m /\ Sorted R n)
RlistIsMonotone = ... :
  {A|Type}{R:Rel A A}(trans R)->{b,c|A}{m|list A}(R c b)->
  (Rlist R b m)->Rlist R c m
PermPreservesRlist = ... :
  {A|Type}{R:Rel A A}{c|A}{m,n|list A}(Perm m n)->(Rlist R c m)->
  Rlist R c n
SortedPermsHaveRHeads = ... :
  {A|Type}{R:Rel A A}(refl R)->{b,c|A}{m,n|list A}
  (Sorted R (cons b m))->(Sorted R (cons c n))->
  (Perm (cons b m) (cons c n))->R b c
SortedPermsHaveEqualHeads = ... :
```

```
    {A|Type}{R:Rel A A}(refl R)->({a,b|A}(R a b)->(R b a)->Eq a b)->
    {b,c|A}{m,n|list A}(Sorted R (cons b m))->(Sorted R (cons c n))->
    (Perm (cons b m) (cons c n))->Eq b c
  SortedPermsAreEqual = ... :
    {A|Type}{R:Rel A A}(refl R)->({a,b|A}(R a b)->(R b a)->Eq a b)->
    {l,m|list A}(Sorted R l)->(Sorted R m)->(Perm l m)->Eq l m
```

13 Finite Sets

The finite set datatype is a family of types parameterised by nat, where the type `fin n` has exactly n elements, which can be thought of as the set of numbers $\{0, \ldots, n-1\}$.

The function `f_plus` lifts elements from $\{0, \ldots, n{-}1\}$ to $\{0, \ldots, m{+}n{-}1\}$, `f_plus_assoc` coerces elements in $\{0, \ldots, (m+(n+o))-1\}$ to $\{0, \ldots, ((m+n)+o))-1\}$.

```
 ** Module lib_finite Imports lib_nat
 $fin : nat->Type(fin)
 $f_zero : {n:nat}fin (suc n)
 $f_suc : {n|nat}(fin n)->fin (suc n)
 $fin_elim :
    {C_fin:{n|nat}(fin n)->TYPE}({n:nat}C_fin (f_zero n))->
    ({n|nat}{x:fin n}(C_fin x)->C_fin (f_suc x))->{n|nat}{z:fin n}
    C_fin z
[[C_fin:{n|nat}(fin n)->TYPE][f_f_zero:{n1:nat}C_fin (f_zero n1)]
 [f_f_suc:{n|nat}{x:fin n}(C_fin x)->C_fin (f_suc x)][n1:nat][n|nat]
 [x:fin n]
    fin_elim C_fin f_f_zero f_f_suc (f_zero n1)  ==>  f_f_zero n1
 || fin_elim C_fin f_f_zero f_f_suc (f_suc x)  ==>
    f_f_suc x (fin_elim C_fin f_f_zero f_f_suc x)]

  f_plus = ... : {n,m|nat}(fin n)->fin (plus m n)
  fin_elim'_lemma = ... :
    {n:nat}{m:fin (suc n)}{P:(fin (suc n))->Type}(P (f_zero n))->
    ({m'5:fin n}P (f_suc m'5))->P m
  fin_elim' = ... :
    {n:nat}{P:(fin (suc n))->Type}(P (f_zero n))->
    ({m:fin n}P (f_suc m))->{m:fin (suc n)}P m
  fin_rec = ... :
    {C|Type}(nat->C)->({n|nat}(fin n)->C->C)->{n|nat}(fin n)->C
  fin_iter = ... : {C|Type}(nat->C)->(nat->C->C)->{n|nat}(fin n)->C
  fin_ind = ... :
    {P:{n:nat}(fin n)->Prop}({n:nat}P (f_zero n))->
    ({n|nat}{x:fin n}(P x)->P (f_suc x))->{n|nat}{z:fin n}P z
```

```
fin_rec' = ... : {n:nat}{C|Type}C->((fin n)->C)->(fin (suc n))->C
fin_ind' = ... :
  {n:nat}{P:(fin (suc n))->Prop}(P (f_zero n))->
  ({m:fin n}P (f_suc m))->{m:fin (suc n)}P m
peano2_fin = ... : {n:nat}{x:fin n}not (Eq (f_suc x) (f_zero n))
peano3_fin = ... : {n:nat}{x,y:fin n}(Eq (f_suc x) (f_suc y))->Eq x y
inj_void = ... :
  {n:nat}{x:fin (suc n)}(not (Eq x (f_zero n)))->
  Ex ([x':fin n]Eq x (f_suc x'))
finite_is_zero_or_succ = ... :
  {n:nat}{x:fin (suc n)}
  (Eq x (f_zero n) \/ Ex ([x':fin n]Eq x (f_suc x')))
zero_set_is_empty = ... : (fin zero)->absurd
fin_zero_lemma = ... :
  {n|nat}{x:fin n}
  nat_elim ([x'3:nat](fin x'3)->Type)
   ([x'3:fin zero]{P:(fin zero)->Type}P x'3)
   ([n'3:nat][_:(fin n'3)->Type][_:fin (suc n'3)]{t:Type}t->t) n x
fin_elim_zero = ... : {P:(fin zero)->Type}{x:fin zero}P x
f_plus_assoc = ... :
  {n,o,m|nat}(fin (plus m (plus n o)))->fin (plus (plus m n) o)
fin_to_nat = ... : {n|nat}(fin n)->nat
fin_to_nat_resp_suc = ... :
  {n|nat}{x:fin n}Eq (suc (fin_to_nat x)) (fin_to_nat (f_suc x))
```

14 Vectors

Vectors are defined in the module `lib_vectors`. Some basic functions on vectors are provided in the module `lib_vectors_basics`.

14.1 Vector Definitions

Vectors are another datatype parameterised over `nat` but this time `vect n` is the collection of lists of length `n`. With this datatype we can define the (family of) total functions "nth" for any natural number n which take an element of the finite set $\{0, \ldots, n-1\}$ and a list of length n and returns the appropriate element of the list. In this file we prove that the "nth" function defined here is equivalent (modulo the appropriate conversions) to the "nth" function in the list library.

```
** Module lib_vectors Imports lib_finite lib_list_length
$vect : (Type)->nat->Type(vect)
$vnil : {A:Type}vect A zero
```

```
 $vcons : {A:Type}A->{n|nat}(vect A n)->vect A (suc n)
 $vect_elim :
    {A:Type}{C_vect:{n|nat}(vect A n)->Type}(C_vect (vnil A))->
    ({a:A}{n|nat}{v:vect A n}(C_vect v)->C_vect (vcons A a v))->{n|nat}
    {z:vect A n}C_vect z
[[A:Type][C_vect:{n|nat}(vect A n)->Type][f_vnil:C_vect (vnil A)]
 [f_vcons:{a:A}{n|nat}{v:vect A n}(C_vect v)->C_vect (vcons A a v)][a:A]
 [n|nat][v:vect A n]
    vect_elim A C_vect f_vnil f_vcons (vnil A)  ==>  f_vnil
 || vect_elim A C_vect f_vnil f_vcons (vcons A a v)  ==>
    f_vcons a v (vect_elim A C_vect f_vnil f_vcons v)]

  vect_elim_ne = ... :
    {A:Type}{C:{n|nat}(vect A (suc n))->Type}
    ({a:A}{n|nat}{v:vect A n}C (vcons A a v))->{n|nat}{v:vect A (suc n)}
    C v
  vect_elim'_lemma = ... :
    {A:Type}{n:nat}{m:vect A (suc n)}{P:(vect A (suc n))->Type}
    ({a:A}{m'6:vect A n}P (vcons A a m'6))->P m
  vect_rec = ... :
    {A:Type}{C|Type}C->(A->{n|nat}(vect A n)->C->C)->{n|nat}(vect A n)->
    C
  vect_iter = ... :
    {A:Type}{C|Type}C->(nat->A->C->C)->{n|nat}(vect A n)->C
  vect_ind = ... :
    {A:Type}{P:{n:nat}(vect A n)->Prop}(P (vnil A))->
    ({a:A}{n|nat}{v:vect A n}(P v)->P (vcons A a v))->{n|nat}
    {z:vect A n}P z
  vnil_lemma = ... :
    {A:Type}{n:nat}{v:vect A n}
    nat_elim|([x:nat](vect A x)->Prop)
     ([v'4:vect A zero]Eq v'4 (vnil A))
     ([x:nat][_:(vect A x)->Prop][_:vect A (suc x)]trueProp) n v
  vect_zero_eq_vnil = ... : {A:Type}{v:vect A zero}Eq v (vnil A)
  vect_head = ... : {A:Type}{n|nat}(vect A (suc n))->A
  vect_tail = ... : {A:Type}{n|nat}(vect A (suc n))->vect A n
  cons_inj_lemma1 = ... :
    {A:Type}{a,b:A}{n:nat}{v:vect A n}(Eq (vcons A a v) (vcons A b v))->
    Eq a b
  cons_inj_lemma2 = ... :
    {A:Type}{a:A}{n:nat}{v,w:vect A n}(Eq (vcons A a v) (vcons A a w))->
    Eq v w
  vect_head_cons_tail = ... :
```

```
    {A:Type}{n|nat}{v:vect A (suc n)}
    Eq (vcons A (vect_head A v) (vect_tail A v)) v
  vect_nth = ... : {A:Type}{n|nat}(fin n)->(vect A n)->A
  vect_update = ... : {A:Type}{n|nat}(fin n)->(vect A n)->A->vect A n
  vect_character = ... :
    {A:Type}
    [vect_character_rhs=nat_elim ([n:nat](vect A n)->vect A n)
                         ([_:vect A zero]vnil A)
                         ([x1:nat][_:(vect A x1)->vect A x1]
                          [v:vect A (suc x1)]
                          vcons A (vect_head A v) (vect_tail A v))]
    {n|nat}{v:vect A n}Eq v (vect_character_rhs n v)
  vect_zip = ... :
    {A:Type}{n:nat}(A->A->A)->(vect A n)->(vect A n)->vect A n
  vector_to_list = ... : {A:Type}{n|nat}(vect A n)->list A
  vector_to_list_resp_cons = ... :
    {A:Type}{n|nat}{v:vect A n}{a:A}
    Eq (cons|A a (vector_to_list A v)) (vector_to_list A (vcons A a v))
  vect_nth_equiv_nth = ... :
    {A:Type}{n|nat}{x:fin n}{v:vect A n}{a:A}
    Eq (vect_nth A x v) (nth (fin_to_nat x) (vector_to_list A v) a)
```

14.2 Vector Basics

This module defines some basic vector functions – where basic really just means pure vector functions which do not require any other datatype. There is currently only one theorem `vect_nth_append`: Given two vectors t and s with dimensions m and n, respectively, retrieving the xth element in s with $x \in \{0, \ldots, n-1\}$ is equivalent to retrieving the xth element in `vappend` t s.

```
 ** Module lib_vectors_basics Imports lib_vectors
  vsingleton = ... : {T|Type}T->vect T one
  vappend = ... :
    {m,n|nat}{T|Type}(vect T m)->(vect T n)->vect T (plus m n)
  vtack = ... : {m|nat}{T|Type}T->(vect T m)->vect T (suc m)
  vect_nth_append = ... :
    {T|Type}{n|nat}{x:fin n}{s:vect T n}{m|nat}{t:vect T m}
    Eq (vect_nth T (f_plus x) (vappend t s)) (vect_nth T x s)
```

15 Quotient Types

The directory `lib_quotient` contains one file which defines a notion of quotient types. The file defines a new type operator `QU` which takes a type and a relation and produces a new type representing the quotient of the old type by the relation. Elements of the quotient type are given by the operator `QU_class` which takes an element in the underlying type to one in the quotient, i.e. the equivalence class of the element. The assumption `QU_cor` then states that the equivalence classes of related elements are equal. The assumption `QU_it` restricts the function space on quotient types to only those functions which respect the relation used to form the quotient. Then `QU_ind` says that all elements of the quotient are the equivalence class of some element of the original type. The reduction rule allows function application to normalise as expected. These definitions allow one to reason with a non-substitutive equality as if it were substitutive.

Only for the final theorem do we actually need to assume that the relation quotiented by is a equivalence. This theorem also requires the assumption of extensionality of propositions.

```
 ** Module lib_quotient Imports lib_eq_basics
 $ext_prop : {X,Y:Prop}(X->Y)->(Y->X)->Eq X Y
 $QU : {A|Type(0)}(A->A->Prop)->Type(0)
 $QU_class : {A|Type(0)}{R:A->A->Prop}A->QU R
 $QU_cor :
    {A|Type(0)}{R:A->A->Prop}{a1,a2:A}(R a1 a2)->
    Eq (QU_class R a1) (QU_class R a2)
 $QU_it :
    {A|Type(0)}{R:A->A->Prop}{C|Type}{f:A->C}
    ({a1,a2:A}(R a1 a2)->Eq (f a1) (f a2))->(QU R)->C
 $QU_ind :
    {A|Type(0)}{R:A->A->Prop}{P:(QU R)->Prop}({a:A}P (QU_class R a))->
    {q:QU R}P q
[[A|Type(0)][R:A->A->Prop][C|Type][f:A->C]
 [H:{a1,a2:A}(R a1 a2)->Eq (f a1) (f a2)][a:A]
    QU_it R f H (QU_class R a)  ==>  f a]

  QU_xi = ... :
    {A|Type(0)}{R:A->A->Prop}{C|Type}{f,g:A->C}
    {p:{a1,a2:A}(R a1 a2)->Eq (f a1) (f a2)}
    {q:{a1,a2:A}(R a1 a2)->Eq (g a1) (g a2)}({a:A}Eq (f a) (g a))->
    {x:QU R}Eq (QU_it R f p x) (QU_it R g q x)
  QU_it_ind = ... :
    {A|Type(0)}{R:A->A->Prop}{C|Type}{P:C->Prop}{f:A->C}
    {p:{a1,a2:A}(R a1 a2)->Eq (f a1) (f a2)}({a:A}P (f a))->{x:QU R}
    P (QU_it R f p x)
  QU_in_class = ... :
    {A|Type(0)}{R:A->A->Prop}(sym R)->(trans R)->A->(QU R)->Prop
  QU_cor_inv = ... :
    {A|Type(0)}{R:A->A->Prop}(refl R)->(sym R)->(trans R)->{a,b:A}
```

```
    (Eq (QU_class R a) (QU_class R b))->R a b
```

16 Relations

There are two files, `relations.l`, and `relclosure.l`, which give elementary and infinitary lemmas about the algebra of relations. They pre-date the addition of primitive inductive types to LEGO, so use impredicative encodings to get "System F"-like inductive definitions.

In both files, a glaring omission is the failure to show that all the operations on relations introduced are in fact extensional with respect to `EQRel`. Much more could fruitfully be added.

16.1 Definitions

`relations.l` uses the obvious definitions of binary relations and unary predicates to do a little algebra with the equality on relations given by pointwise (extensional) equality, induced by a pre-order given by inclusion. The logical operators `and`, `or` etc lift to relations, and we also have the tensor product `composeRel` and its left and right adjoints across the pre-order, `impliesRel coimpliesRel`, familiar as Hoare/He's weakest pre-specification etc. These may also be considered in the context of predicates, and functions-as-relations, giving `existsfPred` and so on. These are then used to prove sound a variety of obvious algebraic laws, and as an application to define "the transitive interior" of a relation.

```
** Module lib_rel Imports lib_nat
 Pred = ... : (Type)->Type
 Rel = ... : (Type)->(Type)->Type
 preserves = ... : {s|Type}{t|Type}(s->t)->(Rel s s)->(Rel t t)->Prop
 WF = ... : {s|Type}(Rel s s)->Prop
 TWF = ... : {s|Type}(Rel s s)->Prop
 per = ... : {s|Type}(Rel s s)->Prop
 equiv = ... : {s|Type}(Rel s s)->Prop
 preorder = ... : {s|Type}(Rel s s)->Prop
 PreorderEq = ... : {s|Type}{R:Rel s s}(preorder R)->Rel s s
 SubPred = ... : {s|Type}(Pred s)->(Pred s)->Prop
 reflSubPred = ... : {s|Type}refl (SubPred|s)
 transSubPred = ... : {s|Type}trans (SubPred|s)
 EqPred = ... : {s|Type}Rel (Pred s) (Pred s)
 SubRel = ... : {s|Type}{t|Type}(Rel s t)->(Rel s t)->Prop
 reflSubRel = ... : {s|Type}{t|Type}refl (SubRel|s|t)
 transSubRel = ... : {s|Type}{t|Type}trans (SubRel|s|t)
 EqRel = ... : {s|Type}{t|Type}Rel (Rel s t) (Rel s t)
 andPred = ... : {s|Type}(Pred s)->(Pred s)->Pred s
 orPred = ... : {s|Type}(Pred s)->(Pred s)->Pred s
 impliesPred = ... : {s|Type}(Pred s)->(Pred s)->Pred s
```

```
notPred = ... : {s|Type}(Pred s)->Pred s
andPredI = ... :
  {s|Type}{E,F,G|Pred s}(SubPred E F)->(SubPred E G)->
  SubPred E (andPred F G)
andPredE1 = ... :
  {s|Type}{E,F,G|Pred s}(SubPred E (andPred F G))->SubPred E F
andPredE2 = ... :
  {s|Type}{E,F,G|Pred s}(SubPred E (andPred F G))->SubPred E G
orPredE = ... :
  {s|Type}{E,F,G|Pred s}(SubPred F E)->(SubPred G E)->
  SubPred (orPred F G) E
orPredI1 = ... :
  {s|Type}{E,F,G|Pred s}(SubPred E F)->SubPred E (orPred F G)
orPredI2 = ... :
  {s|Type}{E,F,G|Pred s}(SubPred E G)->SubPred E (orPred F G)
impliesPredI = ... :
  {s|Type}{E,F,G|Pred s}(SubPred (andPred F G) E)->
  SubPred F (impliesPred G E)
impliesPredE = ... :
  {s|Type}{E,F,G|Pred s}(SubPred F (impliesPred G E))->
  SubPred (andPred F G) E
existsf = ... : {s|Type}{t|Type}(t->s)->(Pred t)->Pred s
starf = ... : {s|Type}{t|Type}(t->s)->(Pred s)->Pred t
forallf = ... : {s|Type}{t|Type}(t->s)->(Pred t)->Pred s
existsfE = ... :
  {s|Type}{t|Type}{F|Pred s}{H|Pred t}{f|t->s}
  (SubPred (existsf f H) F)->SubPred H (starf f F)
existsfI = ... :
  {s|Type}{t|Type}{F|Pred s}{H|Pred t}{f|t->s}
  (SubPred H (starf f F))->SubPred (existsf f H) F
forallfE = ... :
  {s|Type}{t|Type}{F|Pred s}{H|Pred t}{f|t->s}
  (SubPred F (forallf f H))->SubPred (starf f F) H
forallfI = ... :
  {s|Type}{t|Type}{F|Pred s}{H|Pred t}{f|t->s}
  (SubPred (starf f F) H)->SubPred F (forallf f H)
op = ... : {s|Type}{t|Type}(Rel s t)->Rel t s
notRel = ... : {s|Type}{t|Type}(Rel s t)->Rel s t
andRel = ... : {s|Type}{t|Type}(Rel s t)->(Rel s t)->Rel s t
orRel = ... : {s|Type}{t|Type}(Rel s t)->(Rel s t)->Rel s t
composeRel = ... :
  {s|Type}{t|Type}{u|Type}(Rel s t)->(Rel t u)->Rel s u
impliesRel = ... :
```

```
  {s|Type}{t|Type}{u|Type}(Rel s t)->(Rel s u)->Rel t u
coimpliesRel = ... :
  {s|Type}{t|Type}{u|Type}(Rel t u)->(Rel s u)->Rel s t
andRelI = ... :
  {s|Type}{t|Type}{P,Q,R|Rel s t}(SubRel R P)->(SubRel R Q)->
  SubRel R (andRel P Q)
andRelE1 = ... :
  {s|Type}{t|Type}{P,Q,R|Rel s t}(SubRel R (andRel P Q))->SubRel R P
andRelE2 = ... :
  {s|Type}{t|Type}{P,Q,R|Rel s t}(SubRel R (andRel P Q))->SubRel R Q
orRelE = ... :
  {s|Type}{t|Type}{P,Q,R|Rel s t}(SubRel P R)->(SubRel Q R)->
  SubRel (orRel P Q) R
orRelI1 = ... :
  {s|Type}{t|Type}{P,Q,R|Rel s t}(SubRel R P)->SubRel R (orRel P Q)
orRelI2 = ... :
  {s|Type}{t|Type}{P,Q,R|Rel s t}(SubRel R Q)->SubRel R (orRel P Q)
impliesRelI = ... :
  {s|Type}{t|Type}{u|Type}{R|Rel s t}{S|Rel t u}{T|Rel s u}
  (SubRel (composeRel R S) T)->SubRel S (impliesRel R T)
impliesRelE = ... :
  {s|Type}{t|Type}{u|Type}{R|Rel s t}{S|Rel t u}{T|Rel s u}
  (SubRel S (impliesRel R T))->SubRel (composeRel R S) T
coimpliesRelI = ... :
  {s|Type}{t|Type}{u|Type}{R|Rel s t}{S|Rel t u}{T|Rel s u}
  (SubRel (composeRel R S) T)->SubRel R (coimpliesRel S T)
coimpliesRelE = ... :
  {s|Type}{t|Type}{u|Type}{R|Rel s t}{S|Rel t u}{T|Rel s u}
  (SubRel R (coimpliesRel S T))->SubRel (composeRel R S) T
existsgh = ... :
  {s|Type}{t|Type}{u|Type}{v|Type}(u->v->s)->(u->v->t)->(Rel u v)->
  Rel s t
stargh = ... :
  {s|Type}{t|Type}{u|Type}{v|Type}(u->v->s)->(u->v->t)->(Rel s t)->
  Rel u v
forallgh = ... :
  {s|Type}{t|Type}{u|Type}{v|Type}(u->v->s)->(u->v->t)->(Rel u v)->
  Rel s t
existsghE = ... :
  {s|Type}{t|Type}{u|Type}{v|Type}{P|Rel s t}{U|Rel u v}{g|u->v->s}
  {h|u->v->t}(SubRel (existsgh g h U) P)->SubRel U (stargh g h P)
existsghI = ... :
  {s|Type}{t|Type}{u|Type}{v|Type}{P|Rel s t}{U|Rel u v}{g|u->v->s}
```

```
  {h|u->v->t}(SubRel U (stargh g h P))->SubRel (existsgh g h U) P
forallghE = ... :
  {s|Type}{t|Type}{u|Type}{v|Type}{P|Rel s t}{U|Rel u v}{g|u->v->s}
  {h|u->v->t}(SubRel P (forallgh g h U))->SubRel (stargh g h P) U
forallghI = ... :
  {s|Type}{t|Type}{u|Type}{v|Type}{P|Rel s t}{U|Rel u v}{g|u->v->s}
  {h|u->v->t}(SubRel (stargh g h P) U)->SubRel P (forallgh g h U)
KPred = ... : {s|Type}Prop->Pred s
KRelL = ... : {s|Type}{t|Type}(Pred s)->Rel s t
KRelR = ... : {s|Type}{t|Type}(Pred t)->Rel s t
leftKanPred = ... : {s|Type}{t|Type}(Rel s t)->(Pred s)->Pred t
rightKanPred = ... : {s|Type}{t|Type}(Rel s t)->(Pred s)->Pred t
notPredIE = ... :
  {s|Type}{E,F|Pred s}(SubPred E F)->SubPred (notPred F) (notPred E)
notRelIE = ... :
  {s|Type}{t|Type}{P,Q|Rel s t}(SubRel P Q)->
  SubRel (notRel Q) (notRel P)
univPred = ... : {s|Type}Pred s
univRel = ... : {s|Type}{t|Type}Rel s t
emptyPred = ... : {s|Type}Pred s
emptyRel = ... : {s|Type}{t|Type}Rel s t
univPredI = ... : {s|Type}{F|Pred s}SubPred F (univPred|s)
emptyPredI = ... : {s|Type}{F|Pred s}SubPred (emptyPred|s) F
univRelI = ... : {s|Type}{t|Type}{R|Rel s t}SubRel R (univRel|s|t)
emptyRelI = ... : {s|Type}{t|Type}{R|Rel s t}SubRel (emptyRel|s|t) R
existfunit = ... :
  {s|Type}{t|Type}{H|Pred t}{f|t->s}SubPred H (starf f (existsf f H))
existfcounit = ... :
  {s|Type}{t|Type}{F|Pred s}{f|t->s}SubPred (existsf f (starf f F)) F
forallfunit = ... :
  {s|Type}{t|Type}{F|Pred s}{f|t->s}SubPred F (forallf f (starf f F))
forallfcounit = ... :
  {s|Type}{t|Type}{H|Pred t}{f|t->s}SubPred (starf f (forallf f H)) H
existghunit = ... :
  {s|Type}{t|Type}{u|Type}{v|Type}{U|Rel u v}{g|u->v->s}{h|u->v->t}
  SubRel U (stargh g h (existsgh g h U))
existghcounit = ... :
  {s|Type}{t|Type}{u|Type}{v|Type}{P|Rel s t}{g|u->v->s}{h|u->v->t}
  SubRel (existsgh g h (stargh g h P)) P
forallghunit = ... :
  {s|Type}{t|Type}{u|Type}{v|Type}{P|Rel s t}{g|u->v->s}{h|u->v->t}
  SubRel P (forallgh g h (stargh g h P))
forallghcounit = ... :
```

```
  {s|Type}{t|Type}{u|Type}{v|Type}{U|Rel u v}{g|u->v->s}{h|u->v->t}
  SubRel (stargh g h (forallgh g h U)) U
monotonePred = ... : {s|Type}{t|Type}((Pred s)->Pred t)->Prop
monotoneRel = ... :
  {s|Type}{t|Type}{u|Type}{v|Type}((Rel s t)->Rel u v)->Prop
opMono = ... :
  {s|Type}{t|Type}{P,Q|Rel s t}(SubRel P Q)->SubRel (op P) (op Q)
opInvol = ... : {s|Type}{t|Type}{P|Rel s t}SubRel P (op (op P))
composeRelMonoL = ... :
  {s|Type}{t|Type}{u|Type}{P|Rel s t}
  monotoneRel|t|u|s|u (composeRel P)
composeRelR = ... :
  {s|Type}{t|Type}{u|Type}(Rel t u)->(Rel s t)->Rel s u
composeRelMonoR = ... :
  {s|Type}{t|Type}{u|Type}{Q|Rel t u}
  monotoneRel|s|t|s|u (composeRelR Q)
composeRelEqL1 = ... :
  {s|Type}{t|Type}{P|Rel s t}SubRel (composeRel (Eq|s) P) P
composeRelEqL2 = ... :
  {s|Type}{t|Type}{P|Rel s t}SubRel P (composeRel (Eq|s) P)
composeRelEqR1 = ... :
  {s|Type}{t|Type}{P|Rel s t}SubRel (composeRel P (Eq|t)) P
composeRelEqR2 = ... :
  {s|Type}{t|Type}{P|Rel s t}SubRel P (composeRel P (Eq|t))
composeRelAssoc1 = ... :
  {s|Type}{t|Type}{u|Type}{v|Type}{P|Rel s t}{Q|Rel t u}{R|Rel u v}
  SubRel (composeRel (composeRel P Q) R)
   (composeRel P (composeRel Q R))
composeRelAssoc2 = ... :
  {s|Type}{t|Type}{u|Type}{v|Type}{P|Rel s t}{Q|Rel t u}{R|Rel u v}
  SubRel (composeRel P (composeRel Q R))
   (composeRel (composeRel P Q) R)
opcompose1 = ... :
  {s|Type}{t|Type}{u|Type}{P|Rel s t}{Q|Rel t u}
  SubRel (composeRel (op Q) (op P)) (op (composeRel P Q))
opcompose2 = ... :
  {s|Type}{t|Type}{u|Type}{P|Rel s t}{Q|Rel t u}
  SubRel (op (composeRel P Q)) (composeRel (op Q) (op P))
ClosedL = ... : {s|Type}{t|Type}(Rel s s)->(Rel s t)->Prop
ClosedR = ... : {s|Type}{t|Type}(Rel t t)->(Rel s t)->Prop
symkernelRel = ... : {s|Type}(Rel s s)->Rel s s
prekernelRelR = ... :
  {s|Type}{S:Rel s s}[P=impliesRel S S]
```

```
  (preorder P /\ {Q:Rel s s}(ClosedR Q S)->SubRel Q P)
prekernelRelL = ... :
  {s|Type}{S:Rel s s}[P=coimpliesRel S S]
  (preorder P /\ {Q:Rel s s}(ClosedL Q S)->SubRel Q P)
kernelRel = ... :
  {s|Type}{S:Rel s s}(sym S)->[kerS=symkernelRel S]
  (equiv kerS /\ {Q:Rel s s}(ClosedL Q S)->(ClosedR Q S)->
   SubRel Q kerS)
```

16.2 Relation closures

`relclosure.l` considers closure operators, concretely as monotone operators on the algebra of relations and predicates. The treatment follows Cohn, Universal Algebra, and was started by Rod and Savi, and put in this form by me. We have intersection closed families of relations, and pre-fixed and post-fixed points as special cases (although I think the language of today has these concepts the other way round: maybe I was just confused!), greatest and least such by intersection closure, proof that they are fixed points. For finitary operators, we also have Scott-Tarski: the least fixed point is reached in ω many steps. The notion of finiteness here is given itself inductively as (the closure of) an intersection closed family. In this case we take the Kuratowski-Sierpinski definition of finite subset (generated by: empty subset, singleton and binary union), others should work too (but be careful, as we are working intuitionistically!). Joint fixed points, considered first by Andrews, which require commuting operators, are also discussed, and their extension to commuting families of operators. These are then illustrated, briefly, by showing that the reflexivity, symmetry and transitivity operators commute. By way of illustration, this allows us to compute the equivalence (reflexive-symmetric-transitive) closure of a relation R as the (transitive closure of the (symmetric closure of the (reflexive closure of R))), although this is not shown in full.

```
** Module lib_rel_closure Imports lib_more_induction lib_rel lib_nat_Le
 intersectionRel = ... : {s|Type}{t|Type}(Pred (Rel s t))->Rel s t
 unionRel = ... : {s|Type}{t|Type}(Pred (Rel s t))->Rel s t
 intersectionPred = ... : {s|Type}(Pred (Pred s))->Pred s
 unionPred = ... : {s|Type}(Pred (Pred s))->Pred s
 intersectionRelI = ... :
   {s|Type}{t|Type}{F|Pred (Rel s t)}{R:Rel s t}
   ({S:Rel s t}(F S)->SubRel R S)->SubRel R (intersectionRel F)
 intersectionRelE = ... :
   {s|Type}{t|Type}{F|Pred (Rel s t)}{R:Rel s t}(F R)->
   SubRel (intersectionRel F) R
 unionRelI = ... :
   {s|Type}{t|Type}{F|Pred (Rel s t)}{R:Rel s t}(F R)->
   SubRel R (unionRel F)
```

```
unionRelE = ... :
  {s|Type}{t|Type}{F|Pred (Rel s t)}{R:Rel s t}
  ({S:Rel s t}(F S)->SubRel S R)->SubRel (unionRel F) R
intersectionPredI = ... :
  {s|Type}{G|Pred (Pred s)}{P:Pred s}({Q:Pred s}(G Q)->SubPred P Q)->
  SubPred P (intersectionPred G)
intersectionPredE = ... :
  {s|Type}{G|Pred (Pred s)}{P:Pred s}(G P)->
  SubPred (intersectionPred G) P
unionPredI = ... :
  {s|Type}{G|Pred (Pred s)}{P:Pred s}(G P)->SubPred P (unionPred G)
unionPredE = ... :
  {s|Type}{G|Pred (Pred s)}{P:Pred s}({Q:Pred s}(G Q)->SubPred Q P)->
  SubPred (unionPred G) P
intReversesSubRel = ... :
  {s|Type}{t|Type}{E,F|Pred (Rel s t)}(SubPred E F)->
  SubRel (intersectionRel F) (intersectionRel E)
unionPreservesSubRel = ... :
  {s|Type}{t|Type}{E,F|Pred (Rel s t)}(SubPred E F)->
  SubRel (unionRel E) (unionRel F)
intReversesSubPred = ... :
  {s|Type}{G,H|Pred (Pred s)}(SubPred G H)->
  SubPred (intersectionPred H) (intersectionPred G)
unionPreservesSubPred = ... :
  {s|Type}{G,H|Pred (Pred s)}(SubPred G H)->
  SubPred (unionPred G) (unionPred H)
IntersectionClosed = ... : {s|Type}{t|Type}(Pred (Rel s t))->Prop
UnionClosed = ... : {s|Type}{t|Type}(Pred (Rel s t))->Prop
intFhasF = ... :
  {s|Type}{t|Type}{F:Pred (Rel s t)}(IntersectionClosed F)->
  F (intersectionRel F)
unionFhasF = ... :
  {s|Type}{t|Type}{F:Pred (Rel s t)}(UnionClosed F)->F (unionRel F)
iclAnd = ... :
  {s|Type}{t|Type}{F:Pred (Rel s t)}(IntersectionClosed F)->
  {G:Pred (Rel s t)}(IntersectionClosed G)->
  IntersectionClosed (andPred F G)
uclAnd = ... :
  {s|Type}{t|Type}{F:Pred (Rel s t)}(UnionClosed F)->
  {G:Pred (Rel s t)}(UnionClosed G)->UnionClosed (andPred F G)
iclRefl = ... : {s|Type}IntersectionClosed (refl|s)
iclSym = ... : {s|Type}IntersectionClosed (sym|s)
iclTrans = ... : {s|Type}IntersectionClosed (trans|s)
```

```
iclPreorder = ... : {s|Type}IntersectionClosed (preorder|s)
iclPer = ... : {s|Type}IntersectionClosed (per|s)
iclEquiv = ... : {s|Type}IntersectionClosed (equiv|s)
uclSym = ... : {s|Type}UnionClosed (sym|s)
closure = ... : {s|Type}{t|Type}(Pred (Rel s t))->(Rel s t)->Rel s t
iclSubRelR = ... :
  {s|Type}{t|Type}{R|Rel s t}IntersectionClosed (SubRel R)
Rbounds = ... : {s|Type}{t|Type}(Rel s t)->(Rel s t)->Prop
uclRbounds = ... :
  {s|Type}{t|Type}{R|Rel s t}UnionClosed (Rbounds|s|t|R)
closureInc = ... :
  {s|Type}{t|Type}{F:Pred (Rel s t)}{R|Rel s t}SubRel R (closure F R)
closureI = ... :
  {s|Type}{t|Type}{F:Pred (Rel s t)}(IntersectionClosed F)->
  {R|Rel s t}F (closure F R)
closureE = ... :
  {s|Type}{t|Type}{F:Pred (Rel s t)}{R,S|Rel s t}(F S)->(SubRel R S)->
  SubRel (closure F R) S
closureEderived = ... :
  {s|Type}{t|Type}{F:Pred (Rel s t)}{R,S|Rel s t}
  (F (andRel (closure F R) S))->(SubRel R S)->SubRel (closure F R) S
PreFix = ... : {s|Type}{t|Type}((Rel s t)->Rel s t)->(Rel s t)->Prop
PostFix = ... : {s|Type}{t|Type}((Rel s t)->Rel s t)->(Rel s t)->Prop
Fix = ... : {s|Type}{t|Type}((Rel s t)->Rel s t)->(Rel s t)->Prop
Increasing = ... : {s|Type}{t|Type}((Rel s t)->Rel s t)->Prop
Idempotent = ... : {s|Type}{t|Type}((Rel s t)->Rel s t)->Prop
Monotone = ... : {s|Type}{t|Type}((Rel s t)->Rel s t)->Prop
Closure = ... : {s|Type}{t|Type}((Rel s t)->Rel s t)->Prop
pfp0 = ... :
  {s|Type}{t|Type}{phi:(Rel s t)->Rel s t}PreFix phi (emptyRel|s|t)
pfp1 = ... :
  {s|Type}{t|Type}{phi:(Rel s t)->Rel s t}PostFix phi (univRel|s|t)
iclPostFix = ... :
  {s|Type}{t|Type}{phi:(Rel s t)->Rel s t}(Monotone phi)->
  IntersectionClosed (PostFix phi)
uclPreFix = ... :
  {s|Type}{t|Type}{phi:(Rel s t)->Rel s t}(Monotone phi)->
  UnionClosed (PreFix phi)
closureClosure = ... :
  {s|Type}{t|Type}{F:Pred (Rel s t)}(IntersectionClosed F)->
  Closure (closure F)
leastFixPoint = ... : {s|Type}{t|Type}((Rel s t)->Rel s t)->Rel s t
lfplemma = ... :
```

```
  {s|Type}{t|Type}{phi:(Rel s t)->Rel s t}(Monotone phi)->
  Fix phi (leastFixPoint phi)
phipower = ... :
  {s|Type}{t|Type}((Rel s t)->Rel s t)->nat->(Rel s t)->Rel s t
phipower0 = ... : {s|Type}{t|Type}((Rel s t)->Rel s t)->nat->Rel s t
phipower1 = ... :
  {s|Type}{t|Type}{phi:(Rel s t)->Rel s t}(Monotone phi)->{R|Rel s t}
  {k|nat}(PreFix phi R)->PreFix phi (phipower phi k R)
phipower2 = ... :
  {s|Type}{t|Type}{phi:(Rel s t)->Rel s t}(Monotone phi)->{m|nat}
  PreFix phi (phipower0 phi m)
phipower3 = ... :
  {s|Type}{t|Type}{phi:(Rel s t)->Rel s t}(Monotone phi)->{m,k|nat}
  SubRel (phipower0 phi m) (phipower0 phi (plus k m))
phipower4 = ... :
  {s|Type}{t|Type}{phi:(Rel s t)->Rel s t}(Monotone phi)->{m,k|nat}
  SubRel (phipower phi k (phipower0 phi m)) (phipower0 phi (plus k m))
phipowersincrease = ... :
  {s|Type}{t|Type}{phi:(Rel s t)->Rel s t}(Monotone phi)->{m,n|nat}
  (Le m n)->SubRel (phipower0 phi m) (phipower0 phi n)
leastPreFixPoint = ... : {s|Type}{t|Type}((Rel s t)->Rel s t)->Rel s t
lpfplemma1 = ... :
  {s|Type}{t|Type}{phi:(Rel s t)->Rel s t}{n|nat}
  SubRel (phipower0 phi n) (leastPreFixPoint phi)
lpfplemma = ... :
  {s|Type}{t|Type}{phi:(Rel s t)->Rel s t}(Monotone phi)->
  PreFix phi (leastPreFixPoint phi)
lfplemma1 = ... :
  {s|Type}{t|Type}{phi:(Rel s t)->Rel s t}(Monotone phi)->{n|nat}
  SubRel (phipower0 phi n) (leastFixPoint phi)
ScottTarski1 = ... :
  {s|Type}{t|Type}{phi:(Rel s t)->Rel s t}(Monotone phi)->
  SubRel (leastPreFixPoint phi) (leastFixPoint phi)
singletonPred = ... : {s|Type}s->Pred s
singletonRel = ... : {s|Type}{t|Type}s->t->Rel s t
finitePred = ... : {s|Type}Pred (Pred s)
finiteRel = ... : {s|Type}{t|Type}Pred (Rel s t)
finitephipowersbound = ... :
  {s|Type}{t|Type}{phi:(Rel s t)->Rel s t}(Monotone phi)->{Xf|Rel s t}
  (finiteRel Xf)->(SubRel Xf (leastPreFixPoint phi))->
  Ex ([n:nat]SubRel Xf (phipower0 phi n))
Algebraic = ... : {s|Type}{t|Type}((Rel s t)->Rel s t)->Prop
ScottTarski2 = ... :
```

```
  {s|Type}{t|Type}{phi:(Rel s t)->Rel s t}(Monotone phi)->
  (Algebraic phi)->SubRel (leastFixPoint phi) (leastPreFixPoint phi)
ScottTarski = ... :
  {s|Type}{t|Type}{phi:(Rel s t)->Rel s t}(Monotone phi)->
  (Algebraic phi)->Fix phi (leastPreFixPoint phi)
F2 = ... :
  {s|Type}{t|Type}((Rel s t)->Rel s t)->((Rel s t)->Rel s t)->
  Pred (Rel s t)
icl2 = ... :
  {s|Type}{t|Type}{phi:(Rel s t)->Rel s t}(Monotone phi)->
  {psi:(Rel s t)->Rel s t}(Monotone psi)->
  IntersectionClosed (F2 phi psi)
int2 = ... :
  {s|Type}{t|Type}((Rel s t)->Rel s t)->((Rel s t)->Rel s t)->Rel s t
jointpostfixpoint2 = ... :
  {s|Type}{t|Type}{phi:(Rel s t)->Rel s t}(Monotone phi)->
  {psi:(Rel s t)->Rel s t}(Monotone psi)->
  (PostFix phi (int2 phi psi) /\ PostFix psi (int2 phi psi))
Commphipsi = ... :
  {s|Type}{t|Type}((Rel s t)->Rel s t)->((Rel s t)->Rel s t)->
  (Rel s t)->Prop
jointprefixpoint2 = ... :
  {s|Type}{t|Type}{phi:(Rel s t)->Rel s t}(Monotone phi)->
  {psi:(Rel s t)->Rel s t}(Monotone psi)->
  (Commphipsi phi psi (int2 phi psi))->
  (PreFix phi (int2 phi psi) /\ PreFix psi (int2 phi psi))
Ffamily = ... :
  {s|Type}{t|Type}{u|Type}(u->(Rel s t)->Rel s t)->(Rel s t)->Prop
iclfamily = ... :
  {s|Type}{t|Type}{u|Type}{phifamily:u->(Rel s t)->Rel s t}
  ({z:u}Monotone (phifamily z))->
  IntersectionClosed (Ffamily phifamily)
intfamily = ... :
  {s|Type}{t|Type}{u|Type}(u->(Rel s t)->Rel s t)->Rel s t
jointpostfixpoint = ... :
  {s|Type}{t|Type}{u|Type}{phifamily:u->(Rel s t)->Rel s t}
  ({z:u}Monotone (phifamily z))->{z:u}
  PostFix (phifamily z) (intfamily phifamily)
Commfamily = ... :
  {s|Type}{t|Type}{u|Type}(u->(Rel s t)->Rel s t)->(Rel s t)->Prop
jointprefixpoint = ... :
  {s|Type}{t|Type}{u|Type}{phifamily:u->(Rel s t)->Rel s t}
  ({z:u}Monotone (phifamily z))->
```

```
  (Commfamily phifamily (intfamily phifamily))->{z:u}
  PreFix (phifamily z) (intfamily phifamily)
reflop = ... : {s|Type}(Rel s s)->s->s->Prop
symop = ... : {s|Type}(Rel s s)->Rel s s
transop = ... : {s|Type}(Rel s s)->Rel s s
reflopMono = ... : {s|Type}Monotone (reflop|s)
symopMono = ... : {s|Type}Monotone (symop|s)
transopMono = ... : {s|Type}Monotone (transop|s)
commreflsym = ... :
  {s|Type}{R:Rel s s}Commphipsi (reflop|s) (symop|s) R
commsymtrans = ... :
  {s|Type}{R:Rel s s}Commphipsi (symop|s) (transop|s) R
commtransrefl = ... :
  {s|Type}{R:Rel s s}Commphipsi (transop|s) (reflop|s) R
Refl = ... : {s|Type}(Rel s s)->Prop
Sym = ... : {s|Type}(Rel s s)->Prop
Trans = ... : {s|Type}(Rel s s)->Prop
```

17 Set theory

The one file currently in this section contains the version of set theory developed for the work described in Jones (1991). It defines a "set" to be the sigma type consisting of a type plus an equivalence relation over that type. Similarly a "function" between two sets is an arrow between the base types and a proof that the function respects equality. Also defined are (two-place) relations, two-place functions, and subsets (defined as predicates over the base types).

```
** Module lib_sets_ccmj Imports lib_logic
 eq_refl = ... : {A|Type}(A->A->Prop)->Prop
 eq_sym = ... : {A|Type}(A->A->Prop)->Prop
 eq_trans = ... : {A|Type}(A->A->Prop)->Prop
 eq_rel = ... : (Type)->Type
 set = ... : Type
 eq = ... : {A|set}A.1->A.1->Prop
 eq_refl_ax = ... : {A|set}eq_refl|A.1|A.2.1
 eq_sym_ax = ... : {A|set}{x,y|A.1}(eq x y)->eq y x
 eq_trans_ax = ... : {A|set}{x,y,z|A.1}(eq x y)->(eq y z)->eq x z
 pres_eq = ... : {A,B|set}(A.1->B.1)->Prop
 function = ... : set->set->Type
 pres_eq_ax = ... : {A,B|set}{f|function A B}[f'4=f.1]pres_eq|A|B f'4
 pr = ... : {A,B|set}A.1->B.1->A.1#B.1
 cross_prod = ... : set->set->set
```

```
curry = ... : {A,B,C:set}(function|(cross_prod A B)|C)->A.1->B.1->C.1
comp = ... : {A,B,C|set}(function A B)->(function B C)->function A C
pres_l = ... : {A:set}(A.1->A.1->Prop)->Prop
pres_r = ... : {A:set}(A.1->A.1->Prop)->Prop
relation = ... : set->Type
rel_pres_eq_l = ... : {A|set}{R|relation A}pres_l A R.1
rel_pres_eq_r = ... : {A|set}{R|relation A}pres_r A R.1
subset = ... : set->Type
pres_eq2 = ... : {A,B:set}(A.1->A.1->B.1)->Prop
function2 = ... : set->set->Type
pres_eq_l = ... :
  {A,B|set}{f|function2 A B}{x,y,z:A.1}(eq x y)->
  eq (f.1 x z) (f.1 y z)
pres_eq_r = ... :
  {A,B|set}{f|function2 A B}{x,y,z:A.1}(eq x y)->
  eq (f.1 z x) (f.1 z y)
```

18 The Integers

The directory `lib_integers` of the library gives a definition of the integers. In it integers are represented as pairs of natural numbers where the pair (n, m) represents the integer $n - m$.

18.1 Extra lemmas for integers

This file contains some lemmas about the natural numbers which are needed for developing the integers with this particular representation. These lemmas are needed in order to prove cancellation of multiplication laws in the integers and rationals. For example, the first lemma is the (integer) theorem that if the product of two integers is zero, then one or other of the argument is equal to zero. All these lemmas are proved by a double induction on natural numbers.

```
** Module lib_int_nat_lemmas Imports lib_nat_Prop_rels
 weird_thing = ... :
   {x,y,a,b:nat}
   (Eq (plus (times x a) (times y b)) (plus (times x b) (times y a)))->
   (Eq x y \/ Eq a b)
 weird_Lt = ... :
   {x,y,a,b:nat}(Lt x y)->(Lt a b)->
   Lt (plus (times x b) (times y a)) (plus (times x a) (times y b))
 weird_Le = ... :
   {x,y,a,b:nat}(Le x y)->(Le a b)->
```

```
  Le (plus (times x b) (times y a)) (plus (times x a) (times y b))
weird_thing_Le = ... :
  {x,y,a,b:nat}
  (Le (plus (times x a) (times y b)) (plus (times x b) (times y a)))->
  (Lt x y)->Le b a
weird_thing_Lt = ... :
  {x,y,a,b:nat}
  (Lt (plus (times x a) (times y b)) (plus (times x b) (times y a)))->
  (Lt x y)->Lt b a
```

18.2 The Integers—Definitions

In this file are given some basic definitions for the integers. The datatype itself, its equality, various constants and some operations are defined. The internal sigma-type is used to define the pairs of integers. This is mainly because using inductive cross-product is slower to run.

```
** Module lib_int_basics Imports lib_int_nat_lemmas lib_nat_minus_thms
    lib_sets_ccmj
 zed = ... : Type
 Eq_zed = ... : zed->zed->Prop
 Eq_refl_zed = ... : refl Eq_zed
 Eq_sym_zed = ... : sym Eq_zed
 Eq_trans_zed = ... : trans Eq_zed
 int = ... : set
 zero_zed = ... : int.1
 one_zed = ... : int.1
 two_zed = ... : int.1
 three_zed = ... : int.1
 four_zed = ... : int.1
 minus_one_zed = ... : int.1
 plus_zed = ... : int.1->int.1->int.1
 times_zed = ... : int.1->int.1->int.1
 minus_zed = ... : int.1->int.1->int.1
 neg_zed = ... : int.1->int.1
```

18.3 Basic Lemmas

This file contains some basic theorems about the integers involving the algebraic operators, e.g. commutativity, associativity etc. Note this file takes quite a long time to load.

The variable `eq` is a generic equality for any set and is defined in the file in the directory `lib_sets`. Here it is always implicitly specialised to `Eq_zed` which is the equality defined on the set `int` in the file `lib_int_basics`.

```
** Module lib_int_basics_thms Imports lib_int_basics
 times_one_zed = ... : {m|int.1}eq m (times_zed one_zed m)
 middle_four_plus_zed = ... :
   {a,b,c,d|int.1}
   eq (plus_zed (plus_zed a c) (plus_zed b d))
    (plus_zed (plus_zed a b) (plus_zed c d))
 times_plus_dist_r_zed = ... :
   {a,b,c|int.1}
   eq (times_zed (plus_zed a b) c)
    (plus_zed (times_zed a c) (times_zed b c))
 times_plus_dist_l_zed = ... :
   {a,b,c|int.1}
   eq (times_zed c (plus_zed a b))
    (plus_zed (times_zed c a) (times_zed c b))
 times_resp_eq_l = ... :
   {a,b,x:int.1}(eq a b)->eq (times_zed x a) (times_zed x b)
 plus_commutes_zed = ... : {p,q|int.1}Eq (plus_zed p q) (plus_zed q p)
 times_commutes_zed = ... :
   {p,q|int.1}Eq (times_zed p q) (times_zed q p)
 times_assoc_zed = ... :
   {a,b,c|int.1}
   eq (times_zed (times_zed a b) c) (times_zed a (times_zed b c))
 plus_resp_eq = ... :
   {a,b,x,y:int.1}(eq a b)->(eq x y)->eq (plus_zed a x) (plus_zed b y)
 minus_resp_eq = ... :
   {a,b,x,y:int.1}(eq a b)->(eq x y)->
   eq (minus_zed a x) (minus_zed b y)
 times_resp_eq_r = ... :
   {a,b,x:int.1}(eq a b)->eq (times_zed a x) (times_zed b x)
 times_resp_eq = ... :
   {a,b,x,y:int.1}(eq a b)->(eq x y)->
   eq (times_zed a x) (times_zed b y)
 neg_resp_eq = ... : {x,y:int.1}(eq x y)->eq (neg_zed x) (neg_zed y)
 neg_minus = ... :
   {x,y:int.1}eq (plus_zed x (neg_zed y)) (minus_zed x y)
 times_cancel_left_zed = ... :
   {x,y,a:int.1}(not (eq a zero_zed))->
   (eq (times_zed a x) (times_zed a y))->eq x y
 middle_four_times_zed = ... :
   {x,y,a,b:int.1}
   eq (times_zed (times_zed x y) (times_zed a b))
    (times_zed (times_zed x a) (times_zed y b))
 eq_zero_zed_imp = ... : {x:zed}(eq zero_zed x)->Eq x.1 x.2
```

```
 imp_eq_zero_zed = ... : {x:zed}(Eq x.1 x.2)->eq zero_zed x
 times_zero_lemma_zed = ... :
   {x,y:zed}(eq (times_zed x y) zero_zed)->
   (eq zero_zed x \/ eq zero_zed y)
 right_dist_minus_times_zed = ... :
   {a,b,c|int.1}
   eq (times_zed (minus_zed a b) c)
    (minus_zed (times_zed a c) (times_zed b c))
 times_neg = ... :
   {x,y:int.1}eq (times_zed x (neg_zed y)) (times_zed y (neg_zed x))
 times_neg_both = ... :
   {x,y:int.1}eq (times_zed x y) (times_zed (neg_zed x) (neg_zed y))
 plus_assoc_zed = ... :
   {a,b,c|int.1}
   eq (plus_zed (plus_zed a b) c) (plus_zed a (plus_zed b c))
 one_times_zed = ... : {x:int.1}eq (times_zed one_zed x) x
 plus_zero_zed = ... : {x:int.1}eq (plus_zed x zero_zed) x
 minus_zed_is = ... :
   {x,y,z:int.1}iff (eq (minus_zed x y) z) (eq x (plus_zed y z))
 n_minus_n_Eq_zero_zed = ... : {x:int.1}eq (minus_zed x x) zero_zed
 plus_inv_minus_zed = ... :
   {x,y:int.1}eq (plus_zed (minus_zed x y) y) x
 bring_neg_out = ... :
   {x,y:int.1}eq (times_zed (neg_zed x) y) (neg_zed (times_zed x y))
 minus_is_minus_zed = ... :
   {a,b:nat}(Le a b)->eq (minus_zed (b,zero) (a,zero)) (minus b a,zero)
```

18.4 Some theorems about Modulo

This file defines the relation Modulo, and proves a couple of theorems about it.

```
** Module lib_int_mod Imports lib_int_basics_thms
 Mod = ... : int.1->int.1->int.1->Prop
 plus_resp_mod = ... :
   {m,x1,x2,r1,r2|int.1}(Mod m x1 r1)->(Mod m x2 r2)->
   Mod m (plus_zed x1 x2) (plus_zed r1 r2)
 times_resp_Mod = ... :
   {m,x1,x2,r1,r2|int.1}(Mod m x1 r1)->(Mod m x2 r2)->
   Mod m (times_zed x1 x2) (times_zed r1 r2)
```

18.5 Boolean valued functions on the integers

This file defines equality, strict and non-strict inequality as boolean valued functions. It also defines the boolean valued test for zero and positive and proves some theorems about them.

```
** Module lib_int_bool Imports lib_int_Prop_rels lib_nat_rels
 int_eq_zero = ... : int.1->bool
 int_Le_bool = ... : int.1->int.1->bool
 int_eq_bool = ... : int.1->int.1->bool
 int_Lt_bool = ... : int.1->int.1->bool
 bool_pos = ... : int.1->bool
 bool_pos_true_imp_pos = ... :
   {x:int.1}(Eq (bool_pos x) true)->Le_zed zero_zed x
 bool_pos_false_imp_pos = ... :
   {x:int.1}(Eq (bool_pos x) false)->Lt_zed x zero_zed
 pos_imp_bool_pos_true = ... :
   {x:int.1}(Le_zed zero_zed x)->Eq (bool_pos x) true
 pos_imp_bool_pos_false = ... :
   {x:int.1}(Lt_zed x zero_zed)->Eq (bool_pos x) false
```

18.6 Propositional relations on integers

In this file we define the order relations on integers and prove a number of theorems about them.

```
** Module lib_int_Prop_rels Imports lib_int_basics_thms
    lib_nat_Prop_rels
 Le_zed = ... : int.1->int.1->Prop
 Lt_zed = ... : int.1->int.1->Prop
 Le_resp_eq_l = ... : {a,b,x:int.1}(eq a b)->(Le_zed a x)->Le_zed b x
 Le_resp_eq_r = ... : {a,b,x:int.1}(eq a b)->(Le_zed x a)->Le_zed x b
 Lt_resp_eq_l = ... : {a,b,x:int.1}(eq a b)->(Lt_zed a x)->Lt_zed b x
 Lt_resp_eq_r = ... : {a,b,x:int.1}(eq a b)->(Lt_zed x a)->Lt_zed x b
 prod_pos_is_pos = ... :
   {x,y:zed}(Lt_zed zero_zed x)->(Lt_zed zero_zed y)->
   Lt_zed zero_zed (times_zed x y)
 pos_imp_non_zero = ... :
   {x:int.1}(Lt_zed zero_zed x)->not (eq x zero_zed)
 eq_imp_Le_zed = ... : {x,y:int.1}(eq x y)->Le_zed x y
 Le_resp_times_zed = ... :
   {x,y,z:int.1}(Le_zed x y)->(Le_zed zero_zed z)->
   Le_zed (times_zed z x) (times_zed z y)
 Lt_resp_times_zed = ... :
```

```
  {x,y,z:int.1}(Lt_zed x y)->(Lt_zed zero_zed z)->
  Lt_zed (times_zed z x) (times_zed z y)
Le_times_cancel_left_zed = ... :
  {x,y,a:int.1}(Lt_zed zero_zed a)->
  (Le_zed (times_zed a x) (times_zed a y))->Le_zed x y
Lt_times_cancel_left_zed = ... :
  {x,y,a:int.1}(Lt_zed zero_zed a)->
  (Lt_zed (times_zed a x) (times_zed a y))->Lt_zed x y
Lt_on_neg = ... :
  {x,y:int.1}(Lt_zed x y)->Lt_zed (neg_zed y) (neg_zed x)
Lt_on_neg_rev = ... :
  {x,y:int.1}(Lt_zed (neg_zed y) (neg_zed x))->Lt_zed x y
Le_on_neg = ... :
  {x,y:int.1}(Le_zed x y)->Le_zed (neg_zed y) (neg_zed x)
Le_on_neg_rev = ... :
  {x,y:int.1}(Le_zed (neg_zed y) (neg_zed x))->Le_zed x y
Le_antisym_zed = ... : {x,y:int.1}(Le_zed x y)->(Le_zed y x)->eq y x
Le_trans_zed = ... :
  {x,y,z:int.1}(Le_zed x y)->(Le_zed y z)->Le_zed x z
not_refl_Lt_zed = ... : {x:int.1}not (Lt_zed x x)
Lt_trans_zed = ... :
  {x,y,z:int.1}(Lt_zed x y)->(Lt_zed y z)->Lt_zed x z
Le_Lt_trans_zed = ... :
  {x,y,z:int.1}(Le_zed x y)->(Lt_zed y z)->Lt_zed x z
Lt_Le_trans_zed = ... :
  {x,y,z:int.1}(Lt_zed x y)->(Le_zed y z)->Lt_zed x z
Le_resp_plus_both_zed = ... :
  {x,y,a,b:int.1}(Le_zed x y)->(Le_zed a b)->
  Le_zed (plus_zed x a) (plus_zed y b)
Lt_resp_plus_both_zed = ... :
  {x,y,a,b:int.1}(Lt_zed x y)->(Lt_zed a b)->
  Lt_zed (plus_zed x a) (plus_zed y b)
Lt_resp_plus_zed = ... :
  {x,y,z:int.1}(Lt_zed x y)->Lt_zed (plus_zed z x) (plus_zed z y)
prod_Le_is_pos = ... :
  {x,y:zed}(Le_zed zero_zed x)->(Le_zed zero_zed y)->
  Le_zed zero_zed (times_zed x y)
decide_int = ... : {x,y:int.1}(Le_zed x y \/ Lt_zed y x)
Le_is_Lt_or_Eq_zed = ... :
  {x,y:int.1}(Le_zed x y)->(Lt_zed x y \/ eq x y)
Le_resp_times_zed_neg = ... :
  {x,y,z:int.1}(Le_zed x y)->(Lt_zed z zero_zed)->
  Le_zed (times_zed z y) (times_zed z x)
```

18.7 Max and min on integers

In this section max and min are defined on integers and a number of theorems proved about them.

```
** Module lib_int_max_min Imports lib_int_basics_thms lib_int_Prop_rels
    lib_max_min
 min_zed = ... : int.1->int.1->int.1
 max_zed = ... : int.1->int.1->int.1
 max_resp_eq = ... :
   {a,b,x,y:int.1}(eq a b)->(eq x y)->eq (max_zed a x) (max_zed b y)
 min_resp_eq = ... :
   {a,b,x,y:int.1}(eq a b)->(eq x y)->eq (min_zed a x) (min_zed b y)
 max_Le_left_zed = ... : {x,y:int.1}Le_zed x (max_zed x y)
 max_Le_right_zed = ... : {x,y:int.1}Le_zed y (max_zed x y)
 min_Le_left_zed = ... : {x,y:int.1}Le_zed (min_zed x y) x
 min_Le_right_zed = ... : {x,y:int.1}Le_zed (min_zed x y) y
 max_is_least_zed = ... :
   {x,y,z:int.1}(Le_zed x z)->(Le_zed y z)->Le_zed (max_zed x y) z
 min_is_greatest_zed = ... :
   {x,y,z:int.1}(Le_zed z x)->(Le_zed z y)->Le_zed z (min_zed x y)
 max_resp_plus_zed = ... :
   {x,y,z:int.1}
   eq (max_zed (plus_zed z x) (plus_zed z y))
    (plus_zed z (max_zed x y))
 min_resp_plus_zed = ... :
   {x,y,z:int.1}
   eq (min_zed (plus_zed z x) (plus_zed z y))
    (plus_zed z (min_zed x y))
 Le_min_left_zed = ... : {x,y:int.1}(Le_zed x y)->eq (min_zed x y) x
 Le_min_right_zed = ... : {x,y:int.1}(Le_zed y x)->eq (min_zed x y) y
 Le_max_right_zed = ... : {x,y:int.1}(Le_zed x y)->eq (max_zed x y) y
 Le_max_left_zed = ... : {x,y:int.1}(Le_zed y x)->eq (max_zed x y) x
 max_resp_times_zed = ... :
   {x,y,a:int.1}(Le_zed zero_zed a)->
   eq (max_zed (times_zed a x) (times_zed a y))
    (times_zed a (max_zed x y))
 min_resp_times_zed = ... :
   {x,y,a:int.1}(Le_zed zero_zed a)->
   eq (min_zed (times_zed a x) (times_zed a y))
    (times_zed a (min_zed x y))
 max_sym_zed = ... : {x,y:int.1}eq (max_zed x y) (max_zed y x)
 min_sym_zed = ... : {x,y:int.1}eq (min_zed x y) (min_zed y x)
 min_zed_idemp = ... : {a|zed}Eq_zed (min_zed a a) a
```

19 Specification and development of functional programs

This module contains the basic definitions of the program specification and data refinement framework and the definitions of some basic specification operations. See (Luo 1993)(Luo 1994, Chapter 8) for further details of the framework for data refinement.

```
 ** Module lib_adt_basics Imports lib_sigma lib_setoid
  STR = ... : Type
 $SPEC : Type(SPEC)
 $make_SPEC : {Str:STR}(Str->Prop)->SPEC
 $SPEC_elim :
    {C_SPEC:SPEC->Type}
    ({Str:STR}{Ax:Str->Prop}C_SPEC (make_SPEC Str Ax))->{z:SPEC}C_SPEC z
[[C_SPEC:SPEC->Type]
 [f_make_SPEC:{Str:STR}{Ax:Str->Prop}C_SPEC (make_SPEC Str Ax)][Str:STR]
 [Ax:Str->Prop]
    SPEC_elim C_SPEC f_make_SPEC (make_SPEC Str Ax)  ==>
    f_make_SPEC Str Ax]

  Str = ... : SPEC->STR
  Ax = ... : {z:SPEC}(Str z)->Prop
  Spec = ... : STR->Type
  Model = ... : SPEC->Type(sigma)
  Consistent = ... : SPEC->Prop
  Impl = ... : {S|STR}(Spec S)->(Spec S)->Prop
  Sat = ... : {SP,SP'|SPEC}((Str SP')->Str SP)->Prop
 $IMPL1 : SPEC->SPEC->Type(IMPL1)
 $make_IMPL1 :
    {SP,SP'|SPEC}{refn:(Str SP')->Str SP}(Sat refn)->IMPL1|SP|SP'
 $IMPL1_elim :
    {SP,SP'|SPEC}{C_IMPL1:(IMPL1|SP|SP')->Type}
    ({refn:(Str SP')->Str SP}{sat:Sat refn}
     C_IMPL1 (make_IMPL1 refn sat))->{z:IMPL1|SP|SP'}C_IMPL1 z
[[SP,SP'|SPEC][C_IMPL1:(IMPL1|SP|SP')->Type]
 [f_make_IMPL1:{refn:(Str SP')->Str SP}{sat:Sat refn}
  C_IMPL1 (make_IMPL1 refn sat)][refn:(Str SP')->Str SP][sat:Sat refn]
    IMPL1_elim C_IMPL1 f_make_IMPL1 (make_IMPL1 refn sat)  ==>
    f_make_IMPL1 refn sat]

  refn = ... : {SP,SP'|SPEC}(IMPL1|SP|SP')->(Str SP')->Str SP
  sat = ... : {SP,SP'|SPEC}{z:IMPL1|SP|SP'}Sat (refn z)
  IMPL = ... : SPEC->SPEC->Type(IMPL1)
  Extend = ... :
```

```
   {SP:SPEC}{ext_str:(Str SP)->STR}
   [S'=Sigma (Str SP) ([s:Str SP]ext_str s)](S'->Prop)->SPEC
 Extend_AxOnly = ... : {SP:SPEC}((Str SP)->Prop)->SPEC
 Join = ... : {S|STR}(Spec S)->(Spec S)->SPEC
 Cross = ... : SPEC->SPEC->SPEC
 SigmaCirc = ... : {SP:SPEC}((Str SP)->SPEC)->SPEC
 Pi = ... : {SP:SPEC}((Str SP)->SPEC)->SPEC
 Con = ... : {S,S'|STR}(S'->S)->(Spec S')->Spec S
 Sel = ... : {S,S'|STR}(S'->S)->(Spec S)->Spec S'
 Abs = ... : {S|STR}(S->S->Prop)->(Spec S)->Spec S
 Hide = ... : {S|STR}{SP:SPEC}((Str SP)->S)->SPEC
```

An example of the the abstract datatype of stacks, with axiomatisation, proofs of congruence properties, and implementation as an array is also included.

```
 ** Module lib_adt_array Imports lib_nat lib_adt_basics
 $Str_ARRAY : STR
 $make_Str_ARRAY :
    {Array:Setoid}(dom Array)->(nat->(dom Array)->nat->dom Array)->
    ((dom Array)->nat->nat)->Str_ARRAY
 $Str_ARRAY_elim :
    {C_Str_ARRAY:Str_ARRAY->Type}
    ({Array:Setoid}{newarray:dom Array}
     {assign:nat->(dom Array)->nat->dom Array}
     {access:(dom Array)->nat->nat}
     C_Str_ARRAY (make_Str_ARRAY Array newarray assign access))->
    {z:Str_ARRAY}C_Str_ARRAY z
[[C_Str_ARRAY:Str_ARRAY->Type]
 [f_make_Str_ARRAY:{Array:Setoid}{newarray:dom Array}
  {assign:nat->(dom Array)->nat->dom Array}
  {access:(dom Array)->nat->nat}
  C_Str_ARRAY (make_Str_ARRAY Array newarray assign access)]
 [Array:Setoid][newarray:dom Array]
 [assign:nat->(dom Array)->nat->dom Array][access:(dom Array)->nat->nat]
    Str_ARRAY_elim C_Str_ARRAY f_make_Str_ARRAY
     (make_Str_ARRAY Array newarray assign access)  ==>
    f_make_Str_ARRAY Array newarray assign access]

  Array = ... : Str_ARRAY->Setoid
  newarray = ... : {z:Str_ARRAY}dom (Array z)
  assign = ... : {z:Str_ARRAY}nat->(dom (Array z))->nat->dom (Array z)
  access = ... : {z:Str_ARRAY}(dom (Array z))->nat->nat
  EqARRAY = ... : Str_ARRAY->Prop
```

```
 AxARRAY1 = ... : Str_ARRAY->Prop
 AxARRAY2 = ... : Str_ARRAY->Prop
 Ax_ARRAY = ... : Str_ARRAY->Prop
 ARRAY = ... : SPEC

** Module lib_adt_stack Imports lib_nat lib_adt_basics
 eq = ... : {S|Setoid}(dom S)->(dom S)->Prop
$Str_STACK : STR
$make_Str_STACK :
   {Stack:Setoid}(dom Stack)->(nat->(dom Stack)->dom Stack)->
   ((dom Stack)->dom Stack)->((dom Stack)->nat)->Str_STACK
$Str_STACK_elim :
   {C_Str_STACK:Str_STACK->Type}
   ({Stack:Setoid}{empty:dom Stack}{push:nat->(dom Stack)->dom Stack}
    {pop:(dom Stack)->dom Stack}{top:(dom Stack)->nat}
    C_Str_STACK (make_Str_STACK Stack empty push pop top))->
   {z:Str_STACK}C_Str_STACK z
[[C_Str_STACK:Str_STACK->Type]
 [f_make_Str_STACK:{Stack:Setoid}{empty:dom Stack}
  {push:nat->(dom Stack)->dom Stack}{pop:(dom Stack)->dom Stack}
  {top:(dom Stack)->nat}
  C_Str_STACK (make_Str_STACK Stack empty push pop top)][Stack:Setoid]
 [empty:dom Stack][push:nat->(dom Stack)->dom Stack]
 [pop:(dom Stack)->dom Stack][top:(dom Stack)->nat]
   Str_STACK_elim C_Str_STACK f_make_Str_STACK
    (make_Str_STACK Stack empty push pop top)  ==>
   f_make_Str_STACK Stack empty push pop top]

 Stack = ... : Str_STACK->Setoid
 empty = ... : {z:Str_STACK}dom (Stack z)
 push = ... : {z:Str_STACK}nat->(dom (Stack z))->dom (Stack z)
 pop = ... : {z:Str_STACK}(dom (Stack z))->dom (Stack z)
 top = ... : {z:Str_STACK}(dom (Stack z))->nat
 EqSTACK = ... : Str_STACK->Prop
 AxSTACK1 = ... : Str_STACK->Prop
 AxSTACK2 = ... : Str_STACK->Prop
 AxSTACK3 = ... : Str_STACK->Prop
 AxSTACK4 = ... : Str_STACK->Prop
 Ax_STACK = ... : Str_STACK->Prop
 STACK = ... : SPEC

** Module lib_adt_stack_cong Imports lib_adt_basics lib_adt_stack
 RespSTACKpush = ... : Str_STACK->Prop
```

```
 RespSTACKpop = ... : Str_STACK->Prop
 RespSTACKtop = ... : Str_STACK->Prop
 RespSTACK = ... : Str_STACK->Prop
 STACK_CONG = ... : SPEC
 iterate = ... : {A|Type}(A->A)->nat->A->A
 pop_iterate = ... :
   {S:Str STACK_CONG}(Ax STACK_CONG S)->{s:dom (Stack S)}{x:nat}
   eq (iterate (pop S) (suc x) s) (iterate (pop S) x (pop S s))
 pop_iterate_resp = ... :
   {S:Str STACK_CONG}(Ax STACK_CONG S)->{s,s':dom (Stack S)}(eq s s')->
   {x:nat}eq (iterate (pop S) x s) (iterate (pop S) x s')
 pop_iterate_empty = ... :
   {S:Str STACK_CONG}(Ax STACK_CONG S)->{i:nat}
   eq (iterate (pop S) i (empty S)) (empty S)
 pop_iterate_push = ... :
   {S:Str STACK_CONG}(Ax STACK_CONG S)->{s:dom (Stack S)}{k,m:nat}
   eq (iterate (pop S) (suc k) (push S m s)) (iterate (pop S) k s)

** Module lib_adt_list_stack Imports lib_list_basics lib_adt_stack_cong
 listStack = ... : Str_STACK
 eqStackL = ... : EqSTACK listStack
 axStack1L = ... : AxSTACK1 listStack
 axStack2L = ... : AxSTACK2 listStack
 axStack3L = ... : AxSTACK3 listStack
 axStack4L = ... : AxSTACK4 listStack
 listStack_correct = ... : Ax_STACK listStack
 respStackL = ... : RespSTACK listStack
 listStack_correct_cong = ... : Ax STACK_CONG listStack

** Module lib_adt_stack_by_array Imports lib_nat_Lt lib_adt_stack
    lib_adt_array
 refnSA = ... : (Str ARRAY)->Str STACK
 equivSTACK = ... : {A:Str ARRAY}Equivalence (eq|(Stack (refnSA A)))
 axSTACK1 = ... : {A:Str ARRAY}AxSTACK1 (refnSA A)
 axSTACK2 = ... : {A:Str ARRAY}AxSTACK2 (refnSA A)
 axSTACK3 = ... : {A:Str ARRAY}(Ax ARRAY A)->AxSTACK3 (refnSA A)
 axSTACK4 = ... : {A:Str ARRAY}(Ax ARRAY A)->AxSTACK4 (refnSA A)
 sat_refnSA = ... : Sat refnSA
```

20 Pure Martin Löf type theory

This section contains modules which have been developed in pure Martin Löf type theory i.e., without using UTT's impredicative extension.

20.1 Basic properties

The module lib_ML contains only one theorem which relates absurd in Prop with empty in SET

```
** Module lib_ML Imports lib_empty lib_unit lib_bool
 absurd_impl_empty = ... : absurd->empty
```

The module lib_nat_Lt_ML contains elimination rules for nat .

```
** Module lib_nat_Lt_ML Imports lib_ML lib_nat_Lt
 complete_elimination = ... :
   {P:nat->Type}({n:nat}({x:nat}(Lt x n)->P x)->P n)->{m:nat}P m
 well_founded_elimination = ... :
   {t|Type}{f:t->nat}{P:t->Type}
   ({x:t}({y:t}(Lt (f y) (f x))->P y)->P x)->{z:t}P z
```

20.2 Natural Numbers: Basic theorems

```
fun nat_diagonal f _ a b = f a b if a and b stem from the same constructor
   |nat_diagonal _ g a b = g a b

fun nat_diagonal_const f _ a b = f if a and b stem from the same constructor
   |nat_diagonal_const g _ _ _ = g

val nat_filter = nat_diagonal_const unit empty

fun nat_Thm 0       0       = unit
   |nat_Thm 0       (suc n) = {n:nat}(Eq 0 (suc n))->empty
   |nat_Thm (suc m) 0       = {m:nat}(Eq (suc m) 0)->empty
   |nat_Thm (suc m) (suc n) = injective suc
```

```
** Module lib_ML_nat Imports lib_empty lib_unit lib_nat
 nat_diagonal = ... : {C|Type}(nat->nat->C)->(nat->nat->C)->nat->nat->C
 nat_diagonal_const = ... : {C|Type}C->C->nat->nat->C
 nat_filter = ... : nat->nat->Type
 f = ... : nat->Type
 g = ... : nat->Type
 nat_Thm = ... : nat->nat->Type
 get_filter = ... : {m,n|nat}(Eq m n)->Type
 nat_thm = ... : {u,v:nat}nat_Thm u v
```

References

Hofmann, M. & Streicher, T. (1994), A groupoid model refutes uniqueness of identity proofs, *in* 'Ninth Annual Symposium on Logic in Computer Science (LICS) (Paris, France)', IEEE, Computer Society Press, pp. 208–212.
*ftp://ftp.dcs.ed.ac.uk/pub/mxh/lics94.ps

Jones, C. (1991), Completing the rationals and metric spaces in LEGO, *in* G. Huet, G. D. Plotkin & C. Jones, eds, 'Proceedings of the Second Workshop on Logical Frameworks (Edinburgh, Scotland)', pp. 209–222.
*http://www.dcs.ed.ac.uk/lfcsinfo/research/types_bra/proc/proc91.ps.gz

Luo, Z. (1993), 'Program specification and data refinement in type theory', *Mathematical Structures in Computer Science* **3**, 333–363.
*http://www.dur.ac.uk/~dcs0zl/MSCS93.dvi

Luo, Z. (1994), *Computation and Reasoning: A Type Theory for Computer Science*, Oxford University Press.

McKinna, J. H. (1992), Delivarables: A Categorical Approach to Program Development in Type Theory, PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh.
*ftp://ftp.dcs.ed.ac.uk/pub/jhm/thesis.ps.gz