Type
`TRANS_LIST : (void -> void)`

Synopsis
Pretty print transitive relations available as the derivation's base relation.

Description
`TRANS_LIST()` pretty print `rel_list`, which is a list of all transitive relations available to be used as the derivation's base relation.

Failure
Never fail.

See also
`ADD_TRANS`, `derivation`, `UNDO_TRANS`

Synopsis
A global variable representing the set of available transitive relations uponwhich a derivation can be based.

Description
`rel_list` is a global assignable variable. The user does not need to be aware of this variable.

It is a list of pairs `(Ri,TRANSi)` where `Ri` is a transitive relation and `TRANSi` is the theorem that justifies this. To do a derivation with respect to a certain relation, it is required that the package *knows* this relation, that is, it is an element of `rel_list`.

`ADD_TRANS` adds a transitive relation to the space of available relations. `UNDO_TRANS` will delete the last added relation.

Failure
Never fail.

See also
`ADD_TRANS`, `derivation`, `TRANS_LIST`, `UNDO_TRANS`

Set_Assumptions

Type
```
Set_Assumptions : ((string # term) list -> void)
```

Synopsis
Set the assumptions of the proof space.

Description
When applied to a list of pairs `(Li,Ai)` it sets each `Ai` as an assumption of the proof space and tags the name `Li` to it. To access the assumptions use `RECALL`.

Failure
Never fail.

See also
`ASSUMPTIONS`, `RECALL`

TRANS_LIST

Type
`RECALL : (string -> thm)`

Synopsis
To access assumptions and lemmas in the proof space.

Description
`RECALL name` returns the theorem labeled `name` from the proof space.

Failure
Fail if `name` is not found in the proof space.

See also
`GEN_RECALL`, `LEMMA`, `Set_Assumptions`.

rel_in_der

Type
`rel_in_der : term`

Synopsis
A global variable representing the relation being considered in the derivation.

Description
`rel_in_der` is a global assignable variable. The user does not need to be aware of this variable.

As a derivation is relevant only with respect to a given transitive relation, say `<`. What is the given relation is stored in variable `rel_in_der` and it is set when one begin a derivation. See `BD` about how to begin a derivation.

Failure
Never fail.

rel_list

Type
`rel_list : (term # thm) list`

Type
`proof_space : (string # thm # string) list`

Synopsis
Global variable representing the abstract proof space.

Description
It is a global assignable variable that represent the abstract proof space. The user does not need to be aware of this representation. Functions are provided to access and manipulate the proof space.

`proof_space` is a list of triples `Lab,Thm,Hint` where `Thm` is either an assumption or a lemma, `Lab` is the name used to access `Thm`, and `Hint` is a piece of hint as how `Thm` is derived.

Failure
Never fail.

See also
`LEMMA`, `PROOF_SPACE`, `Set_Assumptions`, `LU`

PROOF_SPACE

Type
`PROOF_SPACE : (void -> void)`

Synopsis
Pretty print the proof space.

Description
`PROOF_SPACE()` will pretty print the proof space. Each assumption or lemma will be printed with the corresponding name and hint.

Failure
Never fail.

See also
`LEMMA`, `Set_Assumptions`

RECALL

Type
`Lg : ((string # term # string) -> void)`

Synopsis
Set goal (interactive mode) for a new lemma to be added to the proof space.

Description
Let `Asml` be the assumptions of the proof space (it can be viewed using function `ASSUMPTIONS`). `Lg (Lab,Thm,Hint)` set `(Asml,Thm)` as goal. This can then be proven as one would normaly prove a goal. However if one use `Le` instead of `e` to extend the goal, then if it is proven the lemma will be automatically added to the proof space.

Use `Lg` and `Le` to construct the tactic required by `LEMMA`.

Failure
Never fail.

See also
`Le`, `LEMMA`

LU

Type
`LU: (void -> void)`

Synopsis
Undo the last added lemma in the proof space.

Description
`LU()` delete the last added lemma from the proof space. It also prints the remaining proof space.

Failure
Fail if the proof space is empty.

See also
`LEMMA`

proof_space

`Le` is to be used in conjunction with `Lg`. Suppose we want to extend the proof space with a lemma `thm` which is yet to be proved. We want to use the subgoal package to prove this lemma interactively, and when proven automatically add it to the proof space.

Use `Lg` to automatically set the goal complete with the proof space's assumptions. The goal can then be proven as usual. However, one can opt to use `Le Tac` instead of `e Tac` to apply the tactic `Tac` to a given goal. It does the same as `e` except when `thm` is finally proven `Le` will automatically add it to the proof space, complete with its label and hint.

Use `Lg` and `Le` to construct the tactic required by `LEMMA`.

Failure
`Le Tac` fails if `Tac` fails.

See also
`LEMMA`, `Lg`

LEMMA

Type
```
LEMMA : ((string # term # string # tactic) -> void)
```

Synopsis
Extend the proof space with a new lemma.

Description
`LEMMA (Lab,Thm,Hint,Tac)` adds `Thm` as a new lemma in the proof space if the tactic `Tac` can indeed prove `Thm`. The tactic has all the freedom to exploit the whole proof space, which means not only the assumptions but also the previously proven lemmas. If the tactic succeeds, `Thm` will be labeled by name `Lab` and the hint `Hint` is associated to it.

The state of the proof space can be requested at any moment using `PROOF_SPACE` function. To unde the newly added lemma use `LU`.

As the tactic to prove `Thm` may not be trivial, there is an interactive mode to assist the user in producing the tactic. See `Lg` and `Le`.

Failure
Fail if `Tac` cannot prove `Thm`.

See also
`Le`, `Lg`, `LU`, `PROOF_SPACE`, `Set_Assumptions`

Lg

where `x,y,z` are assumed to be all the free variables in `thm`.

Failure
Fail if `name` is not found in the proof space.

See also
`RECALL`, `LEMMA`, `Set_Assumptions`.

GET_TRANS

Type
```
GET_TRANS : conv
```

Synopsis
Get the transitivity theorem of a given relation, provided it is indeed available in the `DERIVATION` package.

Description
Let `V` be the set of all transitive relations made available to be used as the derivation's base relation. `GET_TRANS Rel` will return the transitivity theorem corresponds to `Rel` provided `Rel` is indeed in `V`.

Use `ADD_TRANS` to make a relation 'available'.

Failure
Fail if `Rel` is not 'available'.

See also
`ADD_TRANS`, `derivation`, `TRANS_LIST`

Le

Type
```
Le : (tactic -> void)
```

Synopsis
Expand the goal like `e` except that it adds it to the proof space if proven.

Description

ETD

Type
`ETD : (void -> thm)`

Synopsis
Extract the theorem relating the first and the last expression in the derivation.

Description
A derivation is a list of expression `Ei` where `Ei` is related to `E(i+1)` by either equality or a given transitive relation, say `<`. Then by the transitivity of `<` the derivation implies `E0 < Elast`. `ETD()` returns the theorem that states this.

Failure
Fail if the derivation contains less than two expression. It also fails if there are expressions not related by equlity nor the base relation setup by `BD` at the beginning of the derivation.

See also
`derivation`, `DERIVATION`, `BD`

GEN_RECALL

Type
`GEN_RECALL : (string -> thm)`

Synopsis
To access assumptions and lemmas in the proof space.

Description
Like `RECALL`, `GEN_RECALL name` returns the theorem labeled `name` from the proof space, but with all assumptions discharged and free variables generalized. That is, if `RECALL name` returns

```
A1, A2 |- thm
```

then `GEN_RECALL name` returns

```
|- !x y z. A1 ==> A2 ==> thm
```

Type
`Dg : ((term # term # string) -> void)`

Synopsis
Setup the proper goal to derive a new expression in the derivation.

Description
A derivation is a list of expression `Ei` where `Ei` is related to `E(i+1)` by either equality or a given transitive relation, say `<`.

To extend the derivation with a new expression `Exp` the function `DERIVE` is used. The function requires a tactic `Tac`. Let `ELast` be the last derived expression. As deriving `Exp` essentially means proving

```
?- Rel ELast Exp
```

`Tac` is the tactic to prove it.

`Tac` may not be trivial. To help constructing `Tac` there is an interactive mode. `Dg (Rel,Exp,Hint)` can be used to automatically setup the goal to `?- Rel ELast Exp`.

One is the free to expand the goal using `e`. However, if one use `De` instead then this will automatically add `Exp` to the derivation once all proof obligations are proven.

Failure
Fail if the derivation is empty.

See also
`De`, `DERIVE`

DU

Type
`DU : (void -> void)`

Synopsis
Undo the last derived expression.

Description
`DU()` undo the last expression in the derivation, that is, simply deleting it from the derivation. It also pretty prints the remaining derivation.

Failure
Fail if derivation is empty.

See also
`DERIVATION`, `DERIVE`

Description
`DERIVATION()` simply pretty prints the derivation, complete with the hint.

Failure
Never fail.

See also
`derivation`, `DERIVE`, `DU`

DERIVE

Type
```
DERIVE : ((term # term # string # tactic) -> void)
```

Synopsis
Extend a derivation.

Description
A derivation is a list of expression `Ei` where `Ei` is related to `E(i+1)` by either equality or a given transitive relation, say `<`.

`DERIVE (Rel,Exp,Hint,Tac)` extends the derivation with `Exp` provided `Tac` can prove that `Exp` is indeed derivable from the last derived expression, or with other words if `Tac` can prove

```
?- Rel Elast Exp
```

where `Elast` is the last derived expression. `Hint` is a piece of hint as how `Exp` can be derived.

`Rel` must be either equality or the chosen base relation setup by `BD` at the beginning of the derivation.

As `Tac` may not be trivial, there is an interactive mode to help constructing `Tac`. See `Dg` and `De`.

Failure
Fail if `Tac` fails.

See also
`BD`, `derivation`, `DERIVATION`, `Dg`, `De`, `DU`

Dg

derivation

Type
```
derivation : (term # term # thm # string) list
```

Synopsis
A global variable to represent the abstarct derivation.

Description
`derivation` is a global assignable variable representing abstract derivation. The user does not need to be aware of this variable. There are functions to manipulate and pretty print `derivation`.

A derivation is basically a list of expressions. Each expression is related to the next one by equality or by a given transitive relation, say `<`. By the transitivity of `<` the derivation implies

```
BeginExpr < LastExpr
```

which is the most essential information indeed. In addition however, a derivation also give information of *how* `LastExpr` is derived.

The variable `derivation` is a list of tuples `(Ri,Ei,Ti,Hi)` where `Ei` is the i-th expression in the derivation, *Ri* is the relation between `E(i-1)` and `Ei`, `Ti` is the theorem justifying that indeed `Ri E(i-1) Ei` holds, and `Hi` is a piece of hint as how `Ei` is derived from `E(i-1)`.

As a derivation is relevant only with respect to a given transitive relation, say `<`, so each `Ei` is either `<` or `=`. What is the given relation is stored in variable `rel_in_der` and it is set when one begin a derivation. See `BD` about how to begin a derivation.

A derivation can be extended with `DERIVE` and the last extension can be undo-ed by `DU`.

Failure
Never fail.

See also
`BD`, `DERIVE`, `DU`

DERIVATION

Type
```
DERIVATION : (void -> void)
```

Synopsis
Pretty prints the derivation so far.

Type
`cur_lemma : (string # term # string)`

Synopsis
A global variable used in LEMMA package.

Description
It is a global assignable variable to hold the lemma to be added to the proof space by using interactive LEMMA mode. In interactive mode, a new lemma is proven by setting it as a goal using `Lg` (instead of the standard `g` of the subgoal package). `Lg` stores the lemma to be proven in `cur_lemma`. The goal can be expanded by `Le` instead of the standard `e`. When the goal is proven `Le` automatically add the lemma as stored in `cur_lemma` to the proof space.

`cur_lemma` is a triple `(Lab,Thm,Hint)` where `Thm` is the lemma to be proven, `Lab` is the name by which `Thm` later can be accessed, and `Hint` is a piece of hint as how to derive `Thm`.

Failure
Never fail.

See also
`Lg`, `Le`

De

Type
`De : (tactic -> void)`

Synopsis
Expand a goal with a tactic. Just like `e` but can automatically extend a derivation.

Description
To interactively construct the tactic required by `DERIVE` to extend a derivation, one first setup the proper goal for the new expression using `Dg`. The goal can then be manipulated normally using `e` and `b` from the subgoal package. However using `De` instead of `e` will automatically extend the derivation with the new expression once all proof obligations of the goal setup by `Dg` are proven.

If the new expression is proven, `De` also pretty prints the new state of the derivation.

Failure
`De Tac` fails if `Tac` fails.

See also
`DERIVE`, `Dg`

Description
`BD Rel Exp` starts a derivation. `Exp` is the begin expression and the derivation is with respect to the relation `Rel`. `Rel` is required to be transitive and known by the package. To print all known relations use `TRANS_LIST` and to add new relation use `ADD_TRANS`.

A derivation can have assumptions which have to be set using `Set_Assumptions` and these can be accessed using `RECALL`.

A derivation can be extended using `DERIVE` and printed using `DERIVATION`.

Failure
Never fail.

See also
`DERIVATION`, `DERIVE`, `Set_Assumptions`

cur_expr

Type
```
cur_expr : (term # term # term # string)
```

Synopsis
A global variable representing the new expression to be derived in the interactive derivation mode.

Description
`cur_expr` is a global assignable variable. The user does not need to be aware of this variable.

Let say we have a derivation `E0 > ... > En` and now we want to extend the derivation with `E(n+1)`. The function `DERIVE` will do this and it expects a tactic `Tac` to prove the derivation step.

Proving the step essentially means proving `En > E(n+1)` and to construct the right `Tac` to prove it there is also an interactive derivation mode. In the interactive mode `Lg` will set the proper goal to derive `E(n+1)` from `En`. The goal can further expanded by `Le` instead of `e` and when `E(n+1)` is finally derived, it is then automatically added to the derivation.

`cur_expr` is used by `Lg` to store the `E(n+1)` and used by `Le` to add it to the derivation.

Failure
Never fail.

See also
`Le], [[Lg`

cur_lemma

Synopsis
Global variable representing the abstract assumption space.

Description
It is a global assignable variable representing a subset of the proof space that makes the assumtions of the proof. The user does not need to be aware of this variable. Functions are provided to access and manipulate the assumption space.

Failure
Never fail.

See also
`ASSUMPTIONS`, `Set_Assumptions`, `proof_space`

ASSUMPTIONS

Type
```
ASSUMPTIONS : (void -> void)
```

Synopsis
Pretty print the assumptions of the proof space.

Description
`ASSUMPTIONS()` pretty prints the assumptions of the proof space.

Failure
Never fail.

See also
`Set_Assumptions`

BD

Type
```
BD : (term -> term -> void)
```

Synopsis
Begin a derivation.

Chapter 2

Manual

ADD_TRANS

Type
`ADD_TRANS : ((term # thm) -> void)`

Synopsis
Add a transitive relation to the set of relations available as the derivation's base relation.

Description
A derivation is a list of expression `Ei` where `Ei` is related to `E(i+1)` by either equality or a given transitive relation, say `<`. Then by the transitivity of `<` the derivation implies `E0 < Elast`. This is the most essential information gained from a derivation.

However to be used as a base relation, `<` must first be made known to the package. `ADD_TRANS (Rel,TRANS)` will make relation `Rel` available as the derivation's base relation. `TRANS` is a theorem that justifies the transitivity of `Rel`.

A relation added canbe undo-ed by `UNDO_TRANS`. The transitivity theorem that corresponds to a particular relation can be accessed by `GET_TRANS`. The list of all available relations can be printed using `TRANS_LIST`.

See also
`derivation`, `GET_TRANS`, `TRANS_LIST`, `UNDO_TRANS`

asm_space

Type
`asm_space : string list`

Chapter 1

A Few Notes

DerLem is a tool, built on top of the standard subgoal package, that provides:

1. A layer simulating equational proof style which is frequently used in pen-and-paper proofs. This will make equational proofs look more natural in HOL.

2. A layer to administrate forward proving. This is to complement the subgoal package, which is basically oriented to backward proving.

Features of DerLem include facility to insert comments, naming of assumptions, and pretty printing. In addition, these features will improve documentability of the proofs.

There is an article, *'On the Style of Mechanical Proving'* which accompanies this manual, which should serve as user's manual. There is also some toy examples to help the user to familiarize hilself with the package.

How to Install

Simply copy all `*.ml` files to whatever directory you wish the library to be located.

How to Load

Simply type `loadf 'DerLem'`, assuming you have properly set HOL loading path to include the directory where you have put the package.

Reference Manual of DerLem Package

ISWB Prasetya
Rijkuniversiteit Utrecht, Vakgroep Informatica
Postbus 80.089, 3508 TB Utrecht, Nederland
Email: wishnu@cs.ruu.nl

March 23, 1994