LEGO 1.3 Reference Card

Thomas Schreiber

November 11, 1998

You are strongly encouraged to use the LEGO system together with Emacs. Alternatively, invoke `lego` to start LEGO in a shell. Notice that all commands need to be terminated by a semicolon. `Drop` will enable you to exit LEGO and return to UNIX.

The Syntax of Terms

`Prop`, `Type`	kinds
{**id** : **term**}**term**, {**id** | **term**}**term**, **term->term**	Π abstraction
[**id** : **term**] **term**, [**id** | **term**] **term**	λ abstraction
term term	application
term symb term	infix application
term . **term**	postfix application
<**id** : **term**>**term**, <**id** | **term**>**term**, **term#term**	Σ type
(**term** , **term** , . . .)	tuple
(**term** , **term** , . . . : **term**)	annotated tuple
term.1, **term**.2	projections
(**term** : **term**)	type cast
[**id=term**] **term**	local definition

LEGO State and Proof State

After loading LEGO, the system enters a state called LEGO state, in which you may extend the context. Using the command `Goal` you enter the proof state. The system returns to LEGO state, if a proof is finished. The following commands apply to both states

[**id=term**]	persistent definition
$[**id=term**]	non-persistent definition
`Cd` "**directory**"	change directory
`Configure Equality` id_1 id_2 id_3	set up equality id_1
`Configure Infix` **symb** `[left|right]` **n**	introduce infix operator **symb**
`Configure Qnify` id_1 id_2	equip unification tactic with acyclicity theorem

`Configure Theorems` id_1 id_2 id_3	select method of constructor conflict proof
`Ctxt`, `Ctxt` **n**, `Ctxt` **id**	display the current context
`Decls`, `Decls` **n**, `Decls` **id**	display the declarations in the current context
(`Dnf` **term**)	compute display-normal form of **term**
`Dnf TReg`	compute display-normal form of the current type
`Dnf VReg`	compute display-normal form of the current term
`echo` "**comment**"	print **comment**
`Equiv TReg` **term**	replace current type with equivalent type **term**
`Equiv VReg` **term**	replace current term with equivalent term **term**
`ExpAll TReg` **n**	expand all definitions in the current type to a depth of
`ExpAll VReg` **n**	expand all definitions in the current term to a depth of
`Expand TReg` id_1 id_2 . . .	use definitions id_1 id_2 . . . to expand the current type
`Expand VReg` id_1 id_2 . . .	use definitions id_1 id_2 . . . to expand the current term
`ExportState` "**filename**"	save the current LEGO process in **filename**
`Help`	print help message
(`Hnf` **term**)	compute (weak) head normal form of **term**
(`Hnf` **n term**)	compute (weak) head normal form of **term** for **n** top levels
`Hnf TReg`	compute (weak) head normal form of the current type
`Hnf` **n** `TReg`	compute (weak) head normal form of the current typ for **n** toplevels
`Hnf VReg`	compute (weak) head normal form of the current term
`Hnf` **n** `VReg`	compute (weak) head normal form of the current ter for **n** toplevels
`Init` **theory**,	initialise the context where **theory** is one of `LF`, `PCC`, `XC` `XCC'`, `XCC_s` or `XCC'_s`
`line`	insert a blank line
`Normal TReg`	normalise the current type
`Normal VReg`	normalise the current term
(`Normal` **term**)	Normalise **term**
`Pwd`	print the current working directory

LEGO State

[**id** : **term**]	non-persistent declaration
$[**id** : **term**]	persistent declaration
`Discharge` **id**	Abstract and remove all non-persistent entries in the context back to and including **id**
`DischargeKeep` **id**	Abstract all non-persistent entries in the context back to and including **id** and append them to the context

`Forget` **id**	Remove all entries in the context back to and including **id**
`Goal` **term**, `Goal` **id**:**term**	start refinement proof with goal **term**
`$Goal` **term**, `$Goal` **id**:**term**	start refinement proof with non-persistent goal **term**

Syntax for Inductive Types

`Inductive`	$[\mathbf{T_1}:\mathbf{M_1}]$ ... $[\mathbf{T_m}:\mathbf{M_m}]$	
`Double Inversion NoReductions Relation Theorems`		*(optional)*
`ElimOver` **Universe**		*(optional)*
`Parameters`	$[\mathrm{a_1}:\mathrm{N_1}]$ ... $[\mathrm{a_n}:\mathrm{N_n}]$	*(optional)*
`Constructors`	$[\mathrm{cons_1}:\mathrm{L_1}]$ ... $[\mathrm{cons_o}:\mathrm{L_o}]$	

`Record` [**id**:**kind**]
`Fields` $[\mathrm{cons_1}:\mathrm{L_1}]$ ... $[\mathrm{cons_o}:\mathrm{L_o}]$;

Proof State

Certain commands in proof state take the number of a goal as argument, which may be either an absolute value, or a value relative to the number of the current goal. We denote this by [+|-]**n**.

`Assumption` [+|-]**n**	close goal by an assumption
`Claim` **term**	create a new goal (lemma) in a proof
`Cut` $[\mathrm{id_1}=\mathrm{id_2}]$	apply substitution lemma
`Dnf`	compute the display-normal form of the current goal
`ExpAll` **n**, `ExpAll` +**n**	expand all definitions in a goal
`Expand` $\mathrm{id_1}\ \mathrm{id_2}\ldots$	use definitions $\mathrm{id_1}\ \mathrm{id_2}\ldots$ to expand the current goal
`Equiv` **term**	replace current goal with equivalent goal **term**
`Hnf`	compute the (weak) head normal form of the current goal
`Hnf` **n**	compute the (weak) head normal form of the current goal for **n** toplevels
`Immed`	remove all goals which unify with some type in the context
`Invert` **term**	invert **term**
`Induction` **id**	induction on premise **id**
`Induction` **n**	induction on **n**th unnamed premise
`Induction` **term**	abstract **term** from the goal, then do induction on it
`intros` $\mathrm{id_1}\ \mathrm{id_2}\ldots$	named Π introduction on the current goal
`intros` [+|-]**n** $\mathrm{id_1}\ \mathrm{id_2}\ldots$	named Π introduction on the goal ?[+|-]**n**
`intros`	Π introduction on the current goal
`intros` [+|-]**n**	Π introduction on a goal
`Intros` $\mathrm{id_1}\ \mathrm{id_2}\ldots$	named Π introduction on head normal form of the current goal

`Intros` [+|-]**n** $\mathrm{id_1}\ \mathrm{id_2}\ldots$	named Π introduction on head normal form of the goal ?[+|-]**n**
`Intros`	Π introduction on head normal form of the current goal
`Intros` [+|-]**n**	Π introduction on head normal form of a goal
`Intros #`	single introduction on Σ types
`KillRef`	kill the current refinement proof
`Next` [+|-]**n**	focus on another goal
`Prf`	show current proof context
`Normal`	normalise the current goal
`qnify` **n**	unify leftmost **n** equational premises
`qnify`	unify all equational premises
`qnify` **term**	unify equation proved by **term**
`Qnify` **n**	head normalise and unify leftmost **n** equational premises
`Qnify`	head normalise and unify all equational premises
`Qnify` **term**	head normalise and unify equation proved by **term**
`Qrepl` **term**	use the type of **term** as a conditional equation to rewrite the current goal
`Refine` **term**	refine the current goal by **term**
`Refine` [+|-]**n** **term**	refine the goal ?[+|-] **n** by **term**
`Save`, `Save` **id**	save persistent proof term
`$Save`, `$Save` **id**	save non-persistent proof term
`Undo` **n**	undo **n** proof steps

The Module mechanism

`ForgetMark` **id**	forget back to the mark **id**
`Load` **id**	load module **id** if not yet loaded
`Make` **id**	load object file corresponding to module **id**
`Marks`	list current marks
`Module` **id**	module header which creates the mark **id**
`Module` $\mathrm{id_0}$ `Import` $\mathrm{id_1}\ \mathrm{id_2}\ldots$	module header with imported modules $\mathrm{id_1}\ \mathrm{id_2}\ldots$ creating the mark $\mathrm{id_0}$
`Reload` **id**	is an abbreviation for `ForgetMark` **id**; `Load` **id**
`Reload` $\mathrm{id_0}$ `From` $\mathrm{id_1}$	is an abbreviation for `ForgetMark` $\mathrm{id_1}$; `Load` $\mathrm{id_0}$

Tacticals

$\mathrm{exprsn_1}$ `Then` $\mathrm{exprsn_2}$	evaluate $\mathrm{exprsn_1}$, if evaluation succeeds, evaluate $\mathrm{exprsn_2}$
$\mathrm{exprsn_1}$ `Else` $\mathrm{exprsn_2}$	evaluate $\mathrm{exprsn_1}$, if evaluation fails, backtrack and evaluate $\mathrm{exprsn_2}$

`Repeat` $\mathbf{exprsn}_1$	evaluate $\mathbf{exprsn}_1$ `Then` $\mathbf{exprsn}_1$ `Then` ...
`For` **n** $\mathbf{exprsn}_1$	evaluate $\mathbf{exprsn}_1$ `Then` $\mathbf{exprsn}_1$ `Then` ... (**n** times)
`Succeed`	this tactical always succeeds
`Fail`	this tactical always fails
`Try` **exprsn**	evaluate **exprsn** and backtrack if evaluation fails