A HOL Quick Reference

Creating Theories

Theory.new_theory *name*	creates a new theory
Theory.export_theory()	writes theory to disk
TotalDefn.Define *term*	function definition
bossLib.Hol_datatype *type-dec*	defines a concrete datatype
EquivType.define_equivalence_type *rec*	type of equivalence classes
Theory.save_thm(*name*,*thm*)	stores theorem
Tactical.prove(*term*,*tactic*)	proves theorem using tactic
Tactical.store_thm(*name*,*term*,*tactic*)	proves and stores theorem

Goal Stack Operations

goalstackLib.g *term*	starts a new goal
goalstackLib.e *tactic*	applies a tactic to the top goal
goalstackLib.b()	undoes previous expansion
goalstackLib.restart()	undoes all expansions
goalstackLib.drop()	abandons the top goal
goalstackLib.dropn *int*	abandons a number of goals
goalstackLib.p()	prints the state of the top goal
goalstackLib.status()	prints the state of all goals
goalstackLib.top_thm()	returns the last theorem proved
goalstackLib.r *int*	rotates sub-goals
goalstackLib.R *int*	rotates proofs

Term Rewriting Tactics

Rewrite.GEN_REWRITE_TAC *conv-op rws* [*thms*]	used to construct bespoke rewriting tactics; applies *conv-op* to the rewriting conversion
Rewrite.PURE_REWRITE_TAC [*thms*]	rewrites goal only using the given theorems
Rewrite.PURE_ONCE_REWRITE_TAC [*thms*]	as above but executes just a single rewrite
Rewrite.REWRITE_TAC [*thms*]	rewrites goal using theorems and some basic rewrites
Rewrite.ONCE_REWRITE_TAC [*thms*]	as above but executes just a single rewrite
Rewrite.PURE_ASM_REWRITE_TAC [*thms*]	rewrites goal only using assumptions and theorems
Rewrite.PURE_ONCE_ASM_REWRITE_TAC [*thms*]	as above but executes just a single rewrite
Rewrite.ASM_REWRITE_TAC [*thms*]	rewrites using assums., theorems and basic rewrites
Rewrite.ONCE_ASM_REWRITE_TAC [*thms*]	as above but executes just a single rewrite

Some Basic Tactics

bossLib.Cases	case analysis on outermost variable
bossLib.Cases_on *term*	case analysis on given term
bossLib.Induct	induct on outermost variable
bossLib.Induct_on *term*	induct on given term
Tactic.STRIP_TAC	splits on outermost connective
Tactic.EXISTS_TAC *term*	gives witness for existential
Tactic.SELECT_ELIM_TAC	eliminates Hilbert choice operator
Tactic.EQ_TAC	reduces boolean equality to implication
Tactic.ASSUME_TAC *thm*	adds an assumption
Tactic.DISJ1_TAC	selects left disjunct
Tactic.DISJ2_TAC	selects right disjunct
bossLib.SPOSE_NOT_THEN *thm-tactic*	starts proof by contradiction

Some Basic Tacticals

Tactical.THEN	applies tactics in sequence
Tactical.THENL	applies list of tactics to sub-goals
Tactical.THEN1	applies the second tactic to first sub-goal
Tactical.ORELSE	applies second tactic only if the first fails
Tactical.REVERSE	reverses the order of sub-goals
Tactical.ALL_TAC	leaves the goal unchanged
Tactical.TRY	do nothing if the tactic fails
Tactical.REPEAT	repeat a tactic until it fails
Tactic.NTAC	apply a tactic some number of times
Tactical.MAP_EVERY	apply a tactic using theorems in a list

Using Assumptions

bossLib.by(*term*,*tactic*)	add assum. using proof
Tactical.ASSUM_LIST [*thms*]	adds list of theorems
Tactical.POP_ASSUM *thm-tactic*	use first assumption
Tactical.POP_ASSUM_LIST *thms-tactic*	use all assumptions
Tactical.PAT_ASSUM *thm-tactic*	use matching assumption
Tactical.FIRST_X_ASSUM *thm-tactic*	use first successful assum.
Tactic.STRIP_ASSUME_TAC *thm*	split and add assumption
Tactic.WEAKEN_TAC *term-pred*	remove assumptions
Tactic.RULE_ASSUM_TAC	apply rule to assumptions
Tactic.IMP_RES_TAC *thm*	resolve *thm* using assums.
Tactic.RES_TAC	mutually resolve assums.
Q.ABBREV_TAC	abbreviate goal's sub-term

Decision Procedures

tautLib.TAUT_TAC	tautology checker
bossLib.DECIDE_TAC	above, plus linear arithmetic
mesonLib.MESON_TAC [*thms*]	first-order prover
BasicProvers.PROVE_TAC [*thms*]	uses Meson
metisLib.METIS_TAC [*thms*]	new first-order prover
bossLib.EVAL_TAC	evaluation tactic
numLib.ARITH_TAC	for Preburger arithmetic
intLib.ARITH_TAC	uses Omega test
intLib.COOPER_TAC	Cooper's algorithm
realLib.REAL_ARITH_TAC	

Simplification Tactics

simpLib.SIMP_TAC *simpset* [*thms*]	simplifies goal using theorems and simplification set
simpLib.ASM_SIMP_TAC *simpset* [*thms*]	as above but also uses the assumptions
simpLib.FULL_SIMP_TAC *simpset* [*thms*]	simplifies the goal and all the assumptions
BasicProvers.RW_TAC *simpset* [*thms*]	more aggressive simplifier; uses type info. & case splits
BasicProvers.SRW_TAC [*ssfrags*][*thms*]	as above but uses a list of *simpset* fragments
simpLib.rewrites [*thms*]	constructs a rewrite fragment
simpLib.mk_simpset [*ssfrag*]	constructs a *simpset* from fragments
simpLib.++(*simpset*,*ssfrag*)	adds a fragment to a *simpset*
simpLib.&&(*simpset*,[*thms*])	adds rewrites to a *simpset*
simpLib.AC *thm thm*	constructs tagged theorem to enable AC simplification

Simplification Sets and Fragments

pureSimps.pure_ss	minimal *simpset* for conditional rewriting
boolSimps.bool_ss	propositional and first-order logic simplifications, plus beta-conversion
bossLib.std_ss	as above + pairs, options, sums, numeral evaluation & eta reduction
bossLib.arith_ss	as above + arithmetic rewrites and decision procedure for linear arithmetic
bossLib.list_ss	a version of the above for the theory of lists
realLib.real_ss	adds some real number evaluation and rewrites to the arithmetic *simpset*
bossLib.srw_ss()	returns 'stateful' *simpset*; has type thms. from loaded theories

bossLib.augment_srw_ss [*ssfrag*]	adds fragments to the 'stateful' *simpset*
BasicProvers.export_rewrites [*names*]	exports named theorems to the 'stateful' *simpset*

boolSimps.CONJ_ss	congruence rule for conjunction
boolSimps.ETA_ss	eta conversion
boolSimps.LET_ss	rewrites out 'let' terms
boolSimps.DNF_ss	converts term to disjunctive-normal-form
pairSimps.PAIR_ss	rewrites for pairs
optionSimps.OPTION_ss	rewrites for options
stringSimps.STRING_ss	rewrites for strings
numSimps.ARITH_ss	arithmetic rewrites and decision procedure
numSimps.ARITH_AC_ss	AC fragment for addition and multiplication
numSimps.REDUCE_ss	reduces ground-term expressions
listSimps.LIST_ss	rewrites for lists
pred_setSimps.SET_SPEC_ss	rewrites for set membership
pred_setSimps.PRED_SET_ss	rewrites for sets

Specialize and Generalize Rules

Thm.SPEC *term*	specializes one variable in the conclusion of a theorem
Drule.SPECL [*terms*]	specializes zero or more variables in the conclusion of a theorem
Drule.SPEC_ALL	specializes the conclusion of a theorem with its own quantified variables
Drule.GSPEC	as above but uses unique variables
Drule.ISPEC *term*	specializes theorem, with type instantiation if necessary
Drule.ISPECL [*terms*]	specializes theorem zero or more times, with type instantiation if necessary
Thm.INST [*term* |-> *term*]	instantiates free variables in a theorem
Thm.GEN *term*	generalizes the conclusion of a theorem
Drule.GENL [*terms*]	generalizes zero or more variables in the conclusion of a theorem
Drule.GEN_ALL	generalizes the conclusion of a theorem over its own free variables

Some Inference Rules

Conv.CONV_RULE *conv*	makes an inference rule from a conversion
Conv.GSYM *thm*	reverses the first equation(s) encountered in a top-down search
Drule.NOT_EQ_SYM *thm*	swaps left-hand and right-hand sides of a negated equation
Thm.CONJUNCT1 *thm*	extracts left conjunct of theorem
Thm.CONJUNCT2 *thm*	extracts right conjunct of theorem
Drule.CONJUNCTS *thm*	recursively splits conjunctions into a list of conjuncts
Drule.MATCH_MP *thm thm*	Modus Ponens inference rule with automatic matching
Thm.EQ_MP *thm thm*	equality version of the Modus Ponens rule
Thm.EQ_IMP_RULE *thm*	derives forward and backward implication from equality of boolean terms

Some Conversions

bossLib.DECIDE	prove term using a tautology checker and linear arithmetic
Rewrite.REWRITE_CONV [*thms*]	rewrites term using basic rewrites and given theorems
simpLib.SIMP_CONV *simpset* [*thms*]	simplifies term using *simpset* and theorems
computeLib.CBV_CONV *compset*	call-by-value conversion
numLib.num_CONV	equates a non-zero numeral with the form SUC x for some x
numLib.REDUCE_CONV	evaluates arithmetic and boolean ground expressions
numLib.SUC_TO_NUMERAL_DEFN_CONV	translates SUC x equations to use numeral constructors
numLib.EXISTS_LEAST_CONV	when applied to a term $\exists n.P(n)$, this conversion returns: $\vdash (\exists n.P(n)) = \exists n.P(n) \wedge \forall n'.n' < n \Rightarrow \neg P(n')$

Conv.SYM_CONV	interchanges the left and right-hand sides of an equation
Conv.SKOLEM_CONV	proves the existence of a Skolem function
Drule.GEN_ALPHA_CONV	renames the bound variable of an abstraction, quantified term, *etc.*
Thm.BETA_CONV	performs a single step of beta-conversion
Thm.ETA_CONV	performs a top level eta-conversion
PairRules.GEN_PALPHA_CONV	paired variable version of the above
PairRules.PBETA_CONV	paired variable version of the above
PairRules.PETA_CONV	paired variable version of the above

Quantification Conversions

<table>
<tr><td>Conv.SWAP_VARS_CONV</td><td>swaps two universally quantified variables</td></tr>
<tr><td>Conv.SWAP_EXISTS_CONV</td><td>swaps two existentially quantified variables</td></tr>
<tr><td>Conv.{NOT|AND|OR}_{EXISTS|FORALL}_CONV</td><td>moves operation inwards through quantifier</td></tr>
<tr><td>Conv.{EXISTS|FORALL}_{NOT|AND|OR|IMP}_CONV</td><td>moves quantifier inwards through operation</td></tr>
<tr><td>Conv.{LEFT|RIGHT}_{AND|OR|IMP}_{EXISTS|FORALL}_CONV</td><td>moves quantifier of left/right operand outward</td></tr>
</table>

Conversion Operations

Conv.DEPTH_CONV	applies conversion repeatedly to all sub-terms, in bottom-up order
Conv.REDEPTH_CONV	applies conversion bottom-up to sub-terms, retraversing changed ones
Conv.ONCE_DEPTH_CONV	applies conversion once to the first suitable sub-term in top-down order
Conv.TOP_DEPTH_CONV	applies conversion top-down to all sub-terms, retraversing changed ones
Conv.LAND_CONV	applies conversion to the left-hand argument of a binary operator
Conv.RAND_CONV	applies conversion to the operand of an application
Conv.RATOR_CONV	applies conversion to the operator of an application
Conv.BINOP_CONV	applies conversion to both arguments of a binary operator
Conv.LHS_CONV	applies conversion to the left-hand side of an equality
Conv.RHS_CONV	applies conversion to the right-hand side of an equality
Conv.STRIP_QUANT_CONV	applies conversion underneath a quantifier prefix
Conv.STRIP_BINDER_CONV	applies conversion underneath a binder prefix
Conv.FORK_CONV(*conv*,*conv*)	applies a pair of conversions to the arguments of a binary operator
Conv.THENC(*conv*,*conv*)	applies two conversions in sequence
Conv.ORELSEC(*conv*,*conv*)	applies the first of two conversions that succeeds

Parsing

numLib.prefer_num()	give numerals and operators natural number types by default
intLib.prefer_int()	give numerals and operators integer types by default
Parse.overload_on(*name*,*term*)	establishes constant as one of the overloading possibilities for a string
Parse.add_infix(*name*,*int*,*assoc*)	adds string as infix with given precedence & associativity to grammar
Parse.set_fixity *name fixity*	allows the fixity of tokens to be updated
Parse.type_abbrev(*name*,*type*)	establishes a type abbreviation
Parse.add_rule *record*	adds a parsing/printing rule to the global grammar

The Database

DB.match [*names*] *term*	attempt to find matching theorems in the specified theories
DB.find *string*	search for theory element by name fragment
DB.axioms *name*	all the axioms stored in the named theory
DB.theorems *name*	all the theorems stored in the named theory
DB.definitions *name*	all the definitions stored in the named theory
DB.export_theory_as_docfiles *name*	produce *.doc* files for the named theory
DB.html_theory *name*	produce web-page for the named theory

Tracing

Feedback.traces()	returns a list of registered tracing variables
Feedback.set_trace *name int*	set a tracing level for a registered trace
Feedback.reset_trace *name*	resets a tracing variable to its default value
Feedback.reset_traces()	resets all registered tracing variables to their default values
"Rewrite"	tracing variable for term rewriting (0–1)
"Subgoal number"	number of printed sub-goals (10–10000)
"meson"	for the first-order prover (1–2)
"numeral types"	show types of numerals (0–1)
"simplifier"	for the simplifier (0–7)
"types"	printing of types (0–2)
Globals.show_types := *bool*	flag controlling printing of HOL types
Globals.show_assums := *bool*	flag for controlling display of theorem assumptions
Globals.show_tags := *bool*	flag for controlling display of tags in theorem pretty-printer
Lib.start_time()	set a timer running
Lib.end_time *name*	check a running timer, and print out how long it has been running
Lib.time *function*	measure how long a function application takes
Count.thm_count()	returns the current value of the theorem counter
Count.reset_thm_count()	resets the theorem counter
Count.apply *function*	returns the theorem count for a function application