
Specware Quick Reference Documentation

Release 4.2

Kestrel Institute

September 05, 2014

Contents

1	Shell Commands	ii
2	Units (specs, morphisms, diagrams, ...)	ii
3	Names	iii
4	Declarations and Definitions	iii
5	Types	iii
6	Expressions	iv
7	Patterns	iv

1 Shell Commands

Command	Result
help [<i>command</i>]	Print help for shell commands
cd [<i>folder-name</i>]	Change or print current folder
dir dirr	List .sw files in folder (current or recursively)
path [<i>path</i> ;...; <i>path</i>]	Set or print SWPATH environment variable
p[roc] [<i>unit</i>]	Process unit(s)“
cinit	Clear unit cache
show showx [<i>unit</i>]	Process and print unit (normal or extended form)
oblig[ations] [<i>unit</i>]	Print the proof obligations of the unit
punits lpunits [<i>unit</i> [<i>target-file</i>]]	Generate proof-units for unit (global or local)
ctext [<i>spec</i>]	Sets context for evaluation
e[val] eval-lisp [<i>expression</i>]	Evaluate and print expression (directly or in Lisp)
gen-lisp lgen-lisp [<i>spec</i> [<i>target-file</i>]]“	Generate Lisp from spec (global or local)
gen-java [<i>spec</i> [<i>options-spec</i>]]	Generate Java from spec
gen-c [<i>spec</i> [<i>target-file</i>]]	Generate C from spec
make [<i>spec</i>]	Generate C with makefile and call “make” on it
ld cf cl [<i>lisp-file</i>]	Load, compile, or load+compile Lisp file
exit quit	Terminate shell

2 Units (specs, morphisms, diagrams, ...)

Syntax	Construct
[[/]<i>name</i>/...<i>name</i>][#<i>name</i>]	Unit-identifier
<i>unit-id</i> = <i>unit-term</i>	Unit-definition
<i>spec declaration</i> ... endspec	Returns spec-form
<i>qualifier</i> qualifying <i>spec</i>	Qualifies unqualified type- and op-names
translate <i>spec</i> by { [<i>type</i> <i>op</i>] <i>name</i> +-> <i>name</i> , ... }	Spec-translation: replaces lhs names in spec by rhs names
<i>spec</i> [<i>morphism</i>]	Spec-substitution: replaces source spec of morphism by target spec in the given spec
colimit <i>diagram</i>	Returns spec at apex of colimit cocone
obligations <i>spec-or-morphism</i>	Returns spec containing proof obligations
morphism <i>spec</i> -> <i>spec</i> { [type op] <i>name</i> +-> <i>name</i> , ... }	Returns spec-morphism
diagram { <i>diagram-node-or-edge</i> , ... }	Returns diagram
<i>name</i> +-> <i>spec</i>	Diagram-node
<i>name</i> : <i>name</i> -> <i>name</i> +-> <i>morphism</i>	Diagram-edge
generate [c java lisp] <i>spec</i> [in <i>filename</i> with <i>options-spec</i>]	Generates C, Java, or Lisp code prove claim in spec
prove <i>claim</i> in <i>spec</i> [with <i>snark</i>] [using { <i>claim</i> , ... }] [option <i>proveroptions</i>]	Proof-term

3 Names

Syntax	Construct
<i>[qualifier.] name</i>	Type-name, op-name
<i>word-symbol</i>	Qualifier
<i>word-symbol</i> <i>non-word-symbol</i>	Name, constructor, field-name, (type-)var
A3 posNat? z-k	Examples of word-symbols
`~! @\$^ &* - =+ \ : < > / ?	Examples of non-word-symbols
true false	Bool-literal
0 1 ...	Nat-literal
# <i>char-glyph</i> #"	Char-literal
" <i>char-glyph...</i> "	String-literal
A ... Z a ... z 0 ... 9 ! ... # ... \ \ \ " \ a \ b \ t \ n \ v \ f \ r \ s \ x00 ... \ xff	Char-glyph

4 Declarations and Definitions

Syntax	Construct
import <i>spec</i>	Import-declaration
type <i>type-name</i>	Type-declaration
type <i>type-name type-var</i>	Polymorphic type-declaration
type <i>type-name (type-var, ...)</i>	
type <i>type-name [type-var (type-vars)] = type</i>	Type-definition
op <i>op-name [infixl infixr prio] : [[type-var, ...]] type</i>	Op-declaration; optional infix assoc/prio; optional polymorphic type parameters
def <i>[[type-var, ...]] op-name [pattern ...] [: type] = expr</i>	Op-definition; optional polymorphic type parameters; optional formal parameters
axiom theorem conjecture <i>name is [[type-var, ...]] expr</i>	Claim-definition; optional polymorphic type parameters

5 Types

Syntax	Construct
`` * <i>constructor</i> * [* <i>type</i> *] `` ... <i>constructor</i> [<i>type</i>]	Sum type
<i>type</i> -> <i>type</i>	Function type
<i>type</i> * ... * <i>type</i>	Product type
{ <i>field-name</i> : <i>type</i> , ... }	Record type
(<i>type</i> <i>expr</i>)	Subtype (Type-restriction)
{ <i>pattern</i> : <i>type</i> <i>expr</i> }	Subtype (Type-comprehension)
<i>type</i> / <i>expr</i>	Quotient type
<i>type type1 type(type1, ...)</i>	Type-instantiation

6 Expressions

fn <i>[] pattern -> expr ...</i>	Lambda-form
case <i>expr of [] pattern -> expr ...</i>	Case-expression
<i>let</i> <i>pattern = expr in expr</i> let <i>rec-let-binding ... in expr</i>	Let-expression
def <i>name [pattern ...][: type] = expr</i>	Rec-let-binding; optional formal parameters
if <i>expr then expr else expr</i>	If-expression
fa ex <i>(var, ...) expr</i>	Quantification (non-constructive)
<i>expr expr1 ... expr1 op-name expr2</i>	Application (prefix- or infix-application)
<i>expr : type</i>	Annotated-expression
<i>expr . N</i>	Field-selection, product type (N = 1 2 3 ...)
<i>expr . field-name</i>	Field-selection, record type
<i>(expr, expr, ...)</i>	Tuple-display (has product type)
<i>{ field-name = expr, ... }</i>	Record-display (has record type)
<i>[expr, ...]</i>	List-display
project quotient choose <i>expr</i>	Various structors
[embed] <i>constructor</i>	Embedder
embed? <i>constructor</i>	Embedding-test

7 Patterns

Syntax	Construct
<i>pattern : type</i>	Annotated-pattern
<i>var as pattern</i>	Aliased-pattern
<i>patternhd :: patternl</i>	Cons-pattern
<i>constructor [pattern]</i>	Embed-pattern
<i>(pattern , pattern, ...)</i>	Tuple-pattern
<i>{ field-name = pattern , ... }</i>	Record-pattern
<i>[pattern , ...]</i>	List-pattern
<i>pattern expr</i>	Guarded-pattern
<i>_</i>	Wildcard-pattern
<i>var</i>	Variable-pattern
<i>literal</i>	Literal-pattern