
Specware Quick Reference Documentation

Release 4.2

Kestrel Institute

June 24, 2013

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)
c[ontext] [<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> ... <i>endspec</i>	Returns spec-form
<i>qualifier</i> <i>qualifying spec</i>	Qualifies unqualified type- and op-names
<i>translate spec</i> <i>by</i> { [<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
<i>colimit diagram</i>	Returns spec at apex of colimit cocone
<i>obligations spec-or-morphism</i>	Returns spec containing proof obligations
<i>morphism spec</i> -> <i>spec</i> { [<i>type</i> <i>op</i>] <i>name</i> +-> <i>name</i> , ... }	Returns spec-morphism
<i>diagram</i> { <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
<i>generate</i> [<i>c</i> <i>java</i> <i>lisp</i>] <i>spec</i> [<i>in filename</i> <i>with options-spec</i>]	Generates C, Java, or Lisp code prove claim in spec
<i>prove claim</i> <i>in spec</i> [<i>with snark</i>] [<i>using</i> { <i>claim</i> , ... }] [<i>option 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
`` *constructor* [*type*] `` ... <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 typeI type(typeI, ...)</i>	Type-instantiation

6 Expressions

fn [<i>l</i>] <i>pattern</i> -> <i>expr</i> ...	Lambda-form
case <i>expr</i> of [<i>l</i>] <i>pattern</i> -> <i>expr</i> ...	Case-expression
<i>letpattern</i> = <i>expr</i> in <i>expr</i>	Let-expression
let <i>rec-let-binding</i> ... in <i>expr</i>	
def <i>name</i> [<i>pattern</i> ...][: <i>type</i>] = <i>expr</i>	Rec-let-binding; optional formal parameters
if <i>expr</i> then <i>expr</i> else <i>expr</i>	If-expression
fa ex (<i>var</i> , ...) <i>expr</i>	Quantification (non-constructive)
<i>expr</i> <i>expr1</i> ... <i>expr1</i> <i>op-name</i> <i>expr2</i>	Application (prefix- or infix-application)
<i>expr</i> : <i>type</i>	Annotated-expression
<i>expr</i> . <i>N</i>	Field-selection, product type (<i>N</i> = 1 2 3 ...)
<i>expr</i> . <i>field-name</i>	Field-selection, record type
(<i>expr</i> , <i>expr</i> , ...)	Tuple-display (has product type)
{ <i>field-name</i> = <i>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</i> : <i>type</i>	Annotated-pattern
<i>var</i> as <i>pattern</i>	Aliased-pattern
<i>patternhd</i> : : <i>patternil</i>	Cons-pattern
<i>constructor</i> [<i>pattern</i>]	Embed-pattern
(<i>pattern</i> , <i>pattern</i> , ...)	Tuple-pattern
{ <i>field-name</i> = <i>pattern</i> , ... }	Record-pattern
[<i>pattern</i> , ...]	List-pattern
<i>pattern</i> <i>expr</i>	Guarded-pattern
—	Wildcard-pattern
<i>var</i>	Variable-pattern
<i>literal</i>	Literal-pattern