Specware Quick Reference Documentation

Release 4.2

Kestrel Institute

May 29, 2015

Contents

1 Shell Commands

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

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

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

3 Names

Syntax	Construct
[*qualifier*.] *name*	Type-name, op-name
word-symbol	Qualifier
word-symbol | *non-word-symbol*	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
#*Char-glyph*	Char-literal
"*Char-glyph...*"	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 *spec*	Import-declaration
type *type-name*	Type-declaration
type *type-name type-var* **type** *type-name* (*type-var*, ...)	Polymorphic type-declaration
type *type-name* [*type-var* | (*type-vars*)] = *type*	Type-definition
op *op-name* [**infixl** | **infixr** *prio*] : [[*type-var*, ...]] *type*	Op-declaration; optional infix assoc/prio; optional polymorphic type parameters
op [[*type-var*, ...]] *op-name pattern ...* : *type* = *expr*	Op-definition
axiom | **theorem** | **conjecture** *name* **is** [[*type-var*, ...]] *expr*	Claim-definition; optional polymorphic type parameters

5 Types

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

6 Expressions

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

7 Patterns

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