

CSC227: VDM-SL Language Guide

1 Type Definitions

An example of a simple data type definition is:

```
Amount = nat
```

This defines a data type with the name “**Amount**” and states that the values which belong to this type are natural numbers (**nat** is one of the basic types described below).

1.1 Invariants

In VDM-SL it is possible to attach invariants to a type definition.

```
Type name == type expression  
inv pattern == logical expression
```

The *pattern* can be a single identifier representing a typical element of the type or a **mk_** expression if the type is a record.

2 Basic data types and type constructors

Basic types:

Type	Values
nat	Natural numbers
nat1	Natural numbers excl. 0
int	Integers
real	Real numbers
bool	Booleans
char	Characters
token	Tokens

Quote types are written as identifiers surrounded by angle brackets e.g. <Red>.

Type constructors:

Constructor	Description
set of _	Finite sets
seq of _	Finite sequences
map _ to _	Finite mappings
_ _	Type Union
[_]	Optional Type
:: notation	Record Types

3 Data type operators

3.1 The Boolean type

Operator	Name	Type
not b	Negation	bool -> bool
a and b	Conjunction	bool * bool -> bool
a or b	Disjunction	bool * bool -> bool
a => b	Implication	->bool * bool -> bool
a <=> b	Biimplication	bool * bool -> bool

3.2 The Numeric Types

Operator	Name	Type
-x	Unary minus	real -> real
abs x	Absolute value	real -> real
x + y	Sum	real * real -> real
x - y	Difference	real * real -> real
x * y	Product	real * real -> real
x / y	Division	real * real -> real
x**y	Power	real * real -> real
x < y	Less than	real * real -> real
x > y	Greater than	real * real -> real
x <= y	Less or equal	real * real -> real
x >= y	Greater or equal	real * real -> real

3.3 The Character, Quote and Token Types

Characters, quotes and token values can only be compared to each other by equality and inequality.

3.4 Set Types

Set enumeration {e1, e2, ..., en} constructs a set of the enumerated elements. The empty set is represented as {}.

Set comprehension: {e | bd1, bd2, ..., bdm & P} constructs a set by evaluating the expression e on all the bindings for which the predicate P evaluates to **true**. The expression e uses the variables defined in the bindings.

Set range: {e1, ..., e2} where e1 and e2 are numeric expressions denotes the set of integers from e1 *up to* e2 inclusive.

Operator	Name	Type
e in set s1	Membership	set of A -> bool
e not in set s1	Not membership	set of A -> bool
s1 union s2	Union	set of A * set of A -> set of A
s1 inter s2	Intersection	set of A * set of A -> set of A
s1 \ s2	Difference	set of A * set of A -> set of A
s1 subset s2	Subset	set of A * set of A -> bool
card s1	Cardinality	set of A -> nat
dunion ss	Distributed union	set of (set of A) -> set of A
dinter ss	Distributed intersection	set of (set of A) -> set of A

3.5 Sequence Types

Sequence enumeration: $[e_1, e_2, \dots, e_n]$ constructs a sequence of the enumerated elements. The empty sequence is $[]$.

Sequence comprehension: $[e \mid id \text{ in set } S \ \& \ P]$ constructs a sequence by evaluating the expression e on all the bindings for which the predicate P evaluates to **true**. The expression e will use the identifier id . S is a set of numbers and id will be matched to the numbers in the normal order (the smallest number first).

Subsequence: A *subsequence* of a sequence l is a sequence formed from consecutive elements of l ; from n_1 up to and including n_2 . It has the form: $l(n_1, \dots, n_2)$ where n_1 and n_2 are positive integer expressions (less than the length of l).

Operator	Name	Type
hd l	Head	seq of $A \rightarrow A$
tl l	Tail	seq of $A \rightarrow \text{seq of } A$
len l	Length	seq of $A \rightarrow \text{nat}$
elems l	Elements	seq of $A \rightarrow \text{set of } A$
inds l	Indices	seq of $A \rightarrow \text{set of nat1}$
$l_1 \wedge l_2$	Concatenation	seq of $A * \text{seq of } A \rightarrow \text{seq of } A$
conc l_1	Distributed concatenation	seq of (seq of A) \rightarrow seq of A
$l(i)$	Sequence index	seq of $A * \text{nat1} \rightarrow A$

3.6 Mapping Types

Mapping enumeration: $\{a_1 \mapsto b_1, a_2 \mapsto b_2, \dots, a_n \mapsto b_n\}$ constructs a mapping of the enumerated maplets. The empty mapping will be written as $\{\mapsto\}$.

Mapping comprehension: $\{ed \mapsto er \mid bd_1, \dots, bdn \ \& \ P\}$ constructs a mapping by evaluating the expressions ed and er on all the possible bindings for which the predicate P evaluates to **true**. bd_1, \dots, bdn are bindings of free identifiers from the expressions ed and er to sets or types.

Operator	Name	Type
dom m	Domain	map $A \text{ to } B \rightarrow \text{set of } A$
rng m	Range	map $A \text{ to } B \rightarrow \text{set of } B$
$m_1 \text{ munion } m_2$	Map union	map $A \text{ to } B * \text{map } A \text{ to } B \rightarrow \text{map } A \text{ to } B$
$m_1 ++ m_2$	Override	map $A \text{ to } B * \text{map } A \text{ to } B \rightarrow \text{map } A \text{ to } B$
$s <: m$	Domain restrict to	set of $A * \text{map } A \text{ to } B \rightarrow \text{map } A \text{ to } B$
$s <:- m$	Domain restrict by	set of $A * \text{map } A \text{ to } B \rightarrow \text{map } A \text{ to } B$
$m >: s$	Range restrict to	set of $B * \text{map } A \text{ to } B \rightarrow \text{map } A \text{ to } B$
$m >:- s$	Range restrict by	set of $B * \text{map } A \text{ to } B \rightarrow \text{map } A \text{ to } B$
$m(d)$	Mapping apply	map $A \text{ to } B * A \rightarrow B$

3.7 Record Types

Record values are constructed using a record constructor written as $\text{mk_RecId}(a_1, a_2, \dots, a_n)$ where the different a s are arbitrary values and RecId is the name of the record type. Record types are defined as:

Type :: *component name* : *type*
 component name : *type*
 ...
 component name : *type*

For example, for a type defined:

```
Date :: day    : Day
      month  : Month
      year   : Year
```

The record constructor for `Date` is `mk_Date(.,.,.)`.

The field selectors are `_.day`, `_.month` and `_.year`.

3.8 Union and Optional Types

Union types are written as:

```
MasterA = A | B | ...
```

An optional type is written as:

```
[T]
```

This denotes a union between the elements from the type `T` and the special value `nil`.

4 Expressions

A **let expression** has the form:

```
let p1 = e1, ..., pn = en in e
```

where `p1,...,pn` are variables, `e1,...,en` are expressions and `e` is an expression involving `p1,...,pn`.

An **if expression** has the form:

```
if e1 then e2 else e3
```

where `e1` is a Boolean expression, while `e2` and `e3` are expressions of any type.

Quantified Expressions have the form:

Universal: `forall bd1, bd2, ..., bdn & e`

Existential: `exists bd1, bd2, ..., bdn & e`

where each `bdi` is a binding (i.e. either a set binding of the form `pi in set s` or a type binding of the form `pi : type`), and `e` is a Boolean expression involving the bound variables.

5 Function Definition

An explicit function definition has the form:

```
f: A * B * ... * Z -> R
f(a,b,...,z) == expr
pre preexpr(a,b,...,z)
```

An implicit function definition has the form:

```
f(a:A,b:B,...,z:Z) res:R
pre preexpr(a,b,...,z)
post postexpr(a,b,...,z,res)
```