Overture Technical Report Series
No. TR-001

???? 2017

Deleted: September

Deleted: 6

VDM-10 Language Manual

by

Peter Gorm Larsen
Kenneth Lausdahl
Nick Battle
John Fitzgerald
Sune Wolff
Shin Sahara
Marcel Verhoef
Peter W. V. Tran-Jørgensen
Tomohiro Oda
Paul Chisholm

Month	Year	Version	Version of Overture.exe	Comment
April	2010		0.2	
May	2010	1	0.2	
February	2011	2	1.0.0	
July	2012	3	1.2.2	
April	2013	4	2.0.0	
March	2014	5	2.0.4	Includes RMs #16, #17, #18, #20
November	2014	6	2.1.2	Includes RMs #25, #26, #29
August	2015	7	2.3.0	Includes RMs #27
April	2016	8	2.3.4	Review inputs from Paul Chisholm
September	2016	9	2.4.0	RMs #35, #36
????	2017	10	2.5.0	RM #39

ii

Chapter 3
Data Type Definitions

3.1 Basic Data Types

In the following a number of basic types will be presented. Each of them will contain:

- Name of the construct.
- Symbol for the construct.
- Special values belonging to the data type.
- Built-in operators for values belonging to the type.
- Semantics of the built-in operators.
- Examples illustrating how the built-in operators can be used.[1]

For each of the built-in operators the name, the symbol used and the type of the operator will be given together with a description of its semantics (except that the semantics of Equality and Inequality is not described, since it follows the usual semantics). In the semantics description identifiers refer to those used in the corresponding definition of operator type, e.g. a, b, x, y etc.

The basic types are the types defined by the language with distinct values that cannot be analysed into simpler values. There are five fundamental basic types: booleans, numeric types, characters, tokens and quote types.

Deleted: The basic types will be explained one by one in the following.

3.2 Compound Types

3.2.5 Composite Types

Composite types correspond to record types in programming languages. Thus, elements of this type are somewhat similar to the tuples described in the section about product types above. The difference between the record type and the product type is that the different components of a record can be directly selected by means of corresponding selector functions. In addition records are tagged with an identifier which must be used when manipulating the record. The only way to tag a type is by defining it as a record. It is therefore common usage to define records with only one field in order to give it a tag. This is another difference to tuples as a tuple must have at least two entries whereas records can be empty.

[1] In these examples the Meta symbol '≡' will be used to indicate what the given example is equivalent to.

In VDM languages, **is _** is a reserved prefix for names and it is used in an *is expression*. This is a built-in operator which is used to determine which record type a record value belongs to. It is often used to discriminate between the subtypes of a union type and will therefore be explained further in section 3.2.6. In addition to record types the **is _** operator can also determine if a value is of one of the basic types.

In the following this convention will be used: A is a record type, A1, ..., Am are arbitrary types, r, r1, and r2 are record values, i1, ..., im are selectors from the r record value (and these must be unique entrances inside one record definition), e1, ..., em are arbitrary expressions.

Syntax: type = composite type | ... ;

composite type ='**compose**', identifier, '**of**', field list, '**end**' ; field list = { field

} ;

field = [identifier, ':'], type
| [identifier, ':-'], type ;

or the shorthand notation composite type = identifier, '::', field

list ; where identifier denotes both the type name and the tag name.

Equation:

```
A :: selfirst : A1
     selsec   : A2
```

or

```
A :: selfirst :  A1
     selsec   :- A2
```

or

```
A :: A1 A2
```

In the second notation, an *equality abstraction* field is used for the second field selsec. The minus indicates that such a field is ignored when comparing records using the equality operator. In the last

notation the fields of A can only be accessed by pattern matching (like it is done for tuples) as the fields have not been named.

The shorthand notation :: used in the two previous examples where the tag name equals the type name, is the notation most used. The more general **compose** notation is typically used if a composite type has to be specified directly as a component of a more complex type:

```
T = map S to compose A of A1 A2 end
```

It should be noted however that composite types can only be used in type definitions, and not e.g. in signatures to functions or operations.

Typically composite types are used as alternatives in a union type definition (see section 3.2.6) such as:

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

where A and B are defined as composite types themselves. In this situation the **is** predicate

can be used to distinguish the alternatives.

Constructors: The record constructor: **mk**A(a, b) where a belongs to the type A1 and b belongs to the type A2.

The syntax and semantics for all record expressions are given in section 6.11.

Operators:

Operator	Name	Type
r.i	Field select	A * Id → Ai
r1 = r2	Equality	A * A → **bool**
r1 <> r2	Inequality	A * A → **bool**
isA(r1)	Is	Id * MasterA → **bool**

Semantics of Operators:

Operator Name	Semantics Description
Field select	yields the value of the field with fieldname i in the record value r. r must have a field with name i.
Equality[2]	Structural equality over the record. That is, field-by-field equality, recursively applying equality to the constituent fields.

[2] This equality is implicitly provided with the type. It is possible to override the primitive equality – see section 3.4.

Examples: Let Score be defined as

```
Score :: team   : Team
         won    : nat
         drawn  : nat
         lost   : nat
         points : nat;
Team = <Brazil> | <France> | ...
```

and let

sc1 = **mk**Score (<France>, 3, 0, 0, 9), sc2 = **mk**Score (<Denmark>, 1, 1, 1, 4), sc3 = **mk**Score (<SouthAfrica>, 0, 2, 1, 2) and sc4 = **mk**Score (<SaudiArabia>, 0, 1, 2, 1).

Then

sc1.team	≡	<France>	sc4.points	≡	1
			sc2.points > sc3.points	≡	**true**
			isScore(sc4)	≡	**true**
			isbool(sc3)	≡	**false**
			isint(sc1.won)	≡	**true**
			sc4 = sc1	≡	**false**
			sc4 <> sc2	≡	**true**

The equality abstraction field, written using ':-' instead of ':', may be useful, for example, when working with lower level models of an abstract syntax of a programming language. For example, one may wish to add a position information field to a type of identifiers without affecting the true identity of identifiers:

```
Id :: name : seq of char
      pos  :- nat
```

The effect of this will be that the pos field is ignored in equality comparisons, e.g. the following would evaluate to true:

```
mk Id("x",7) = mk Id("x",9)
```

In particular this can be useful when looking up in an environment which is typically modelled as a map of the following form:

```
Env = map Id to Val
```

Such a map will contain at most one index for a specific identifier, and a map lookup will be independent of the pos field.

Moreover, the equality abstraction field will affect set expressions. For example,

```
{mk_Id("x",7),mk_Id("y",8),mk_Id("x",9)}
```

will be equal to

```
{mk_Id("x",?),mk_Id("y",8)}
```

where the question mark stands for 7 or 9.

Finally, note that for equality abstraction fields valid patterns are limited to don't care and identifier patterns. Since equality abstraction fields are ignored when comparing two values, it does not make sense to use more complicated patterns.

3.3 Invariants

If the data types specified by means of equations as described above contain values which should not be allowed, then it is possible to restrict the values in a type by means of an invariant. The result is that the type is restricted to a subset of its original values. Thus, by means of a predicate the acceptable values of the defined type are limited to those where this expression is true. The general scheme for using invariants looks like this:

```
Id = Type
inv pat == expr
```

where pat is a pattern matching the values belonging to the type Id, and expr is a truth-valued expression, involving some or all of the identifiers from the pattern pat. If an invariant is defined, a new (total) function is implicitly created with the signature:

```
inv_Id : Type +> bool
```

This function can be used within other invariant, function or operation definitions.

For instance, recall the record type Score defined on page 25. We can ensure that the number of points awarded is consistent with the number of games won and drawn using an invariant:

```
Score :: team   : Team
         won    : nat
         drawn  : nat
         lost   : nat
         points : nat
inv sc == sc.points = 3* sc.won + sc.drawn;
```

The invariant function implicitly created for this type is:

```
inv_Score : Score +> bool
inv_Score (sc) ==
  sc.points = 3 * sc.won + sc.drawn;
```

3.4 Equality

Every type defined in VDM, both basic and compound types, is provided with an equality relation by default as described earlier. The primitive equality is not always that which is desired. If the values of a data type are normalised then structural equality is adequate, but this is not always the case. Consider for example a data type that represents times and includes time zones. The same point in time is represented differently in different time zones.

A type definition allows an equality relation to be defined explicitly for a type. In such a case the explicit equality relation is employed when comparing values of the type in preference to the primitive equality.

The general scheme for defining an equality relation is:

```
Id = Type
eq pat1 = pat2 == expr
```

or

```
Id :: fields
eq pat1 = pat2 == expr
```

pat1 and pat2 are patterns for two values of the type (or composite type), and expr is a boolean expression that is true exactly when the expressions represented by pat1 and pat2 are equal.

When defined, the explicit equality relation is also employed for inequality comparison with <>.

If an **eq** clause is defined, a new (total) function is created implicitly with the signature:

```
eq_T : T * T +> bool
```

such that `eq_T(t1,t2)` denotes the same value as `t1 = t2`.

Example: Flight matching

```
Flight :: id          : seq1 of char
          departure   : seq1 of char
          depTime     : DateTime
          destination: seq1 of char
eq mk_Flight(i1,d1,dt1,a1) = mk_Flight(i2,d2,dt2,a2) ==
     i1 = i2 and d1 = d2 and a1 = a2 and
     within(dt1, dt2, mk_Minute(10));
```

A simplified definition of a flight consisting of an identifier, departure location, departure date/time, and destination location. Two records refer to the same flight if they have the same identifier, same departure location, same destination location, and a departure time within 10 minutes of each other; it is the last item that renders structural equality inadequate.[3]

Given

f1 = mk_Flight("QF5", "YSSY", '17-04-01 12:20', "WSSS")
f2 = mk_Flight("QF5", "YSSY", '17-04-01 12:28', "WSSS")
f3 = mk_Flight("VOZ42", "YSSY", '16-12-25 02:21', "YBBN")
f4 = mk_Flight("VOZ42", "YSSY", '16-12-24 02:21', "YBBN")
f5 = mk_Flight("VOZ42", "YSSY", '16-12-24 02:21', "YMML")

We have

f1 = f2 ≡ **true**
f3 = f4 ≡ **false**

[3] We assume a type `DateTime`, a type `Minute`, and a function `within`.

f1 = f3 ≡ **false**
f2 <> f4 ≡ **true**
eq_Flight(f4, f5) ≡ **false**

3.5 Order

Numeric types (section 3.1.2) have a primitive order relation. An order relation can be defined explicitly for other types as part of the type definition.

The general scheme for defining an order (less than) relation is:

```
Id = Type
ord pat1 < pat2 == expr
```

or

```
Id :: fields
ord pat1 < pat2 == expr
```

`pat1` and `pat2` are patterns for two values of the type (or composite type), and `expr` is a boolean expression that is true exactly when the expression represented by `pat1` is less than the expression represented by `pat2` in the required order relation.

If an **ord** clause is defined, three new (total) functions are created implicitly with the signatures:

```
ord_T : T * T +> bool
max_T : T * T +> T
min_T : T * T +> T
```

such that

```
ord_T(t1,t2) = t1 < t2
max_T(t1,t2) = t2, if ord_T(t1,t2) or t1 = t2
min_T(t1,t2) = t1, if ord_T(t1,t2) or t1 = t2
```

If an **ord** clause is defined for a type, then the infix operators <. <=, > and >= can be employed with expressions of that type. The equality relation for a type is defined (either explicitly or implicitly), and if the order relation for a type is also defined (explicitly), we have

```
x <= y  <=> x < y or x = y
```

x > y <=> y < x

x >= y <=> x > y **or** x = y

Example: Score revisited

```
Score :: team   : Team
         won    : nat
         drawn  : nat
         lost   : nat
         points : nat
inv sc == sc.points = 3 * sc.won + sc.drawn
ord mk_Score(t1,w1,-,-,p1) < mk_Score(t2,w2,-,-,p2) ==
    p1 < p2 or p1 = p2 and w1 < w2 or
    p1 = p2 and w1 = w2 and t1 < t2;
```

In this case the order is as might be presented in a league table (with greatest element at top):

- Most points first
- If equal on points, most wins first
- Otherwise alphabetic ordering of team name (not defined here)

Given

sc1 = **mk**_Score (<France>, 2, 2, 0, 8)

sc2 = **mk**_Score (<Scotland>, 3, 0, 0, 9)

sc3 = **mk**_Score (<SouthAfrica>, 0, 3, 0, 3)

sc4 = **mk**_Score (<SaudiArabia>, 1, 0, 2, 3).

We have

sc1 < sc2	≡	**true**
sc1 <= sc3	≡	**false**
sc2 > sc3	≡	**true**
sc4 >= sc3	≡	**true**
sc4 < sc3	≡	**false**
ord_Score(sc1, sc2)	≡	**true**

ISBN 978-90-9023705-3

Appendix A

The Syntax of the VDM Languages

A.4 Definitions

A.4.1 Type Definitions

type definitions = **'types'**, [access type definition] ,
{ ';', access type definition }, [';'] ;

access type definition = ([access], [**'static'**]) | ([**'static'**], [access]),
type definition ;

The access part is only possible in VDM++ and VDM-RT.

access = **'public'**
| **'private'**
| **'protected'** ;

type definition =identifier, '=', type, [invariant], [eq clause], [ord clause]
|identifier, '::', field list, [invariant], [eq clause], [ord clause] ;

type = bracketed type
| basic type
| quote type
| composite type
| union type
| product type
| optional type
| set type
| seq type
| map type
| partial function type
| type name
| type variable ; bracketed type = '(', type, ')' ;

basic type = **'bool'** | **'nat'** | **'nat1'** | **'int'** | **'rat'**

| ‘**real**’ | ‘**char**’ | ‘**token**’ ; quote type = quote literal ; composite type =

‘**compose**’, identifier, ‘**of**’, field list, ‘**end**’ ; field list = { field } ;

field = [identifier, ‘:’], type
| [identifier, ‘:-’], type ; union type = type, ‘|’, type, { ‘|’, type }

; product type = type, ‘*’, type, { ‘*’, type } ; optional type = ‘[’, type, ‘]’

;

set type = set0 type
| set1 type ;
set0 type = ‘**set of**’, type ; set1 type = ‘**set1 of**’, type ;

seq type = seq0 type
| seq1 type ; seq0 type = ‘**seq of**’, type ;

seq1 type = ‘**seq1 of**’, type ;

map type = general map type
| injective map type ; general map type = ‘**map**’, type, ‘**to**’,

type ; injective map type = ‘**inmap**’, type, ‘**to**’, type ;

function type = partial function type
| total function type ;

partial function type = discretionary type, ‘->’, type ; total function type =

discretionary type, ‘+>’, type ;

discretionary type = type
| ‘(’, ‘)’ ;

type name = name ;

type variable = type variable identifier ; invariant = ‘**inv**’, invariant initial

function ;

eq clause = ‘**eq**’, pattern, ‘=’, pattern, ‘==’, expression;

ord clause = '**ord**', pattern, '<', pattern, '==', expression;

invariant initial function = pattern, '==', expression ;

Appendix B

Lexical Specification

B.2 Symbols

The following kinds of symbols exist: keywords, delimiters, symbolic literals, and comments. The transformation from characters to symbols is given by the following rules; these use the same notation as the syntax definition but differ in meaning in that no separators may appear between adjacent terminals. Where ambiguity is possible otherwise, two consecutive symbols must be separated by a separator.

keyword = '**#act**' | '**#active**' | '**#fin**' | '**#req**' | '**#waiting**' | '**abs**'
| '**all**' | '**always**' | '**and**' | '**as**' | '**async**' | '**atomic**' | '**be**' |
'**bool**' | '**by**' | '**card**' | '**cases**' | '**char**' | '**class**'
| '**comp**' | '**compose**' | '**conc**' | '**cycles**' | '**dcl**' | '**def**'
| '**definitions**' | '**dinter**' | '**div**' | '**dlmodule**' | '**do**'
| '**dom**' | '**dunion**' | '**duration**' | '**elems**' | '**else**' | '**elseif**'
| '**end**' | '**eq**' | '**error**' | '**errs**' | '**exists**' | '**exists1**' | '**exit**'
| '**exports**' | '**ext**' | '**false**' | '**floor**'
| '**for**' | '**forall**' | '**from**' | '**functions**' | '**hd**' | '**if**' | '**in**'
| '**inds**' | '**inmap**' | '**instance**' | '**int**' | '**inter**'
| '**imports**' | '**init**' | '**inv**' | '**inverse**' | '**iota**' | '**is**'
| '**isofbaseclass**' | '**isofclass**' | '**lambda**' | '**len**' | '**let**'
| '**map**' | '**measure**' | '**merge**' | '**mod**' | '**module**' | '**mu**'
| '**munion**' | '**mutex**' | '**nat**' | '**nat1**' | '**new**' | '**nil**' | '**not**' | '**of**'
| '**operations**' | '**or**' | '**ord**' | '**others**' | '**per**' | '**periodic**' | '**post**'
| '**power**' | '**pre**' | '**private**' | '**protected**' | '**psubset**'
| '**public**' | '**pure**' | '**rat**' | '**rd**' | '**real**' | '**rem**' | '**renamed**'

| ‘**responsibility**’ | ‘**return**’ | ‘**reverse**’ | ‘**rng**’
| ‘**samebaseclass**’ | ‘**sameclass**’ | ‘**self**’ | ‘**seq**’ | ‘**seq1**’
| ‘**set**’ | ‘**set1**’ | ‘**skip**’ | ‘**specified**’ | ‘**sporadic**’ | ‘**st**’ | ‘**start**’
| ‘**startlist**’ | ‘**state**’ | ‘**stop**’ | ‘**stoplist**’
| ‘**struct**’ | ‘**subclass**’ | ‘**subset**’ | ‘**sync**’
| ‘**system**’ | ‘**then**’ | ‘**thread**’ | ‘**threadid**’ | ‘**time**’ | ‘**tixe**’
| ‘**tl**’ | ‘**to**’ | ‘**token**’ | ‘**traces**’ | ‘**trap**’ | ‘**true**’ | ‘**types**’
| ‘**undefined**’ | ‘**union**’ | ‘**uselib**’ | ‘**values**’
| ‘**variables**’ | ‘**while**’ | ‘**with**’ | ‘**wr**’ | ‘**yet**’ | ‘**RESULT**’ ;

identifier = initial letter, { following letter } ;