Dates & Times: An Annotated VDM Specification

by

Paul Chisholm

Overture – Open-source Tools for Formal Modelling

Document History

Version	Date	Description of Changes
1	May 2016	First release

Table of Contents

1 Introduction

A large proportion of automated systems have a fundamental dependence on dates, times, intervals and durations. Air traffic management for example could not be conducted effectively without addressing crucial temporal dependencies that allows prediction of where an aircraft is expected to be at any point in time (the trajectory). The concept of date, time, interval and duration is vital for the precise and accurate delivery of many kinds of services.

In many ways, dates and times are so ubiquitous they are taken for granted, often captured simply with the statement (in the aviation domain) that a system employs Co-ordinated Universal Time (UTC). The programming languages employed to implement systems provide libraries for the manipulation of dates and times. These libraries are not standardised, providing varied application programming interfaces (API), and informal documentation.

This paper presents a formal specification of dates, times, intervals and durations based on International Organization for Standardisation (ISO) 8601 [1], and Request for Comments (RFC) 3339 [2]. The formalism employed is the Vienna Development Method Specification Language (VDM-SL) [3][4][1]. Temporal concepts are not core to VDM-SL. The intent is to provide a standard library that can be used in the specification of any system that requires dates, times, intervals and durations.

2 Overview

In general, we would like specifications to be as high level and abstract as possible; the purpose is to specify the 'what' with no concern for the 'how'. The aim of this specification is rather more fundamental than a typical business focussed problem [5], such as the management of aircraft flight plans. Consequently, the specification is rather low level, and in fact can be viewed as a functional program, but the goal is not to implement, it is to communicate the semantics of dates, times, intervals and durations. The focus of the 'programs' is to present the meaning as clearly as possible; no concern is given to efficiency or concrete representation.

If a specification is limited to a subset of the full VDM-SL language, the specification can be 'simulated' or 'animated'; that is, the functions being specified can be provided with input, and the simulation delivers the result as per the specification. The simulation does not merely compute the result, it verifies all pre-conditions, post-conditions and invariants during simulation. This provides a capability to 'test' the specification and increase confidence in its correctness [6] (also known as model checking).

Due to the low level nature of the date/time specification, it is restricted to the subset of VDM-SL that allows simulation. It is expected this specification will be employed by other specifications that need a model of dates and times, and the authors of those models may want to carry out model simulation. To enable such an approach, it is essential this specification allows simulation.

We employ VDM-SL with modules. The specification focusses on purely functional descriptions; state variables and operations are not employed. The presentation is limited to explicit definitions with decidable constraints[2]. However, where undecidable constraints add value, they are included as comments.

The VDM specification is implemented in the Overture Tool [7] which provides syntax and type checking of the specification, and supports simulation and testing of specifications.

[1] Often the term VDM on its own is used when VDM-SL is intended.
[2] A constraint is decidable if its truth or falsity can be computed automatically in finite time.

3 ISO 8601/RFC 3339

ISO 8601 [1] is the International Organization for Standardization's recommendation for the numeric representation of dates and times. ISO 8601 is a wide ranging and complex specification. RFC 3339 [2] is a profile of ISO 8601 aimed at internet protocols that restricts attention to those aspects of ISO 8601 that are commonly used. This specification is aligned with RFC 3339 for dates and times with the following exceptions:

- A seconds value of 60 (leap second) is not supported;
- RFC 3339 (and ISO 8601) impose no limit on the resolution of times; the finest granularity of this specification is milliseconds.

RFC 3339 does not support time intervals and durations. This specification adopts the following subset of ISO 8601:

- All intervals are represented as a pair of timestamps denoting the start and end of the interval;
- A duration is an abstract entity that is denoted by a number of milliseconds.

4 Commentary

The complete VDM specification as implemented in Overture is presented in appendix A. It is supported by other core libraries:

- Appendix B – specification of character based functionality;
- Appendix C – specification of numeric functionality;
- Appendix D – specification of functionality on sequences (lists);
- Appendix E – specification of functionality on sets.

This section presents a commentary of some aspects of the specifications. It is not the intention to provide an introduction to VDM; refer to the Wikipedia entry [1] for a good introduction. However, we do describe some salient aspects of VDM to assist with the commentary.

4.1 Header

The module header defines:

- The name of the module;
- The modules it imports;
- The types, values and functions exported by the module.

A fragment of that header is:

```
module ISO8601
imports from Numeric all,
        from Set all,
        from Seq all
exports types struct Year
              struct Month
              struct Day
              struct Hour
              struct Minute
              struct Date
              struct Time
              Duration
        values MILLIS_PER_SECOND, SECONDS_PER_MINUTE, MINUTES_PER_HOUR, HOURS_PER_DAY: nat
               DAYS_PER_MONTH, DAYS_PER_MONTH_LEAP: map nat1 to nat1
        functions mkUTC: Hour * Minute * Second +> UTC
```

```
                                    isUTC: Time +> bool
                                    isLeap: Year +> bool
                                    daysInMonth: Year * Month +> nat1
                                    daysInYear: Year +> nat1
                                    dateLess: Date * Date +> bool
                                    dateLeq: Date * Date +> bool
                                    dateGrtr: Date * Date +> bool
                                    dateGeq: Date * Date +> bool
```

Only the types, values and functions defined in the module that are listed in the header are visible to other modules that import this specification. In the case of types, the `struct` qualifier indicates the internal structure of the type is also visible. In modular and object oriented programming languages the internal structure of a type is generally hidden, and manipulation always via functions or methods. However, since types in VDM are typically expressed in an abstract manner, and have associated invariants to preserve integrity of the data, exposing the internal structure of a type generally does not reveal to the client unnecessary implementation detail.

4.2 Data Types

A key part of any specification is the data model that represents the information of interest. VDM allows the expression of data types in a high-level and abstract manner, independent of physical implementation. The notation allows the simple construction of hierarchical and inductive entities based on constituent building blocks. The following is a fragment from the data type section concerning dates:

```
-- A year: 0 = 0AD (or 1BC).
Year = nat
inv year == FIRST_YEAR <= year and year <= LAST_YEAR;

-- A month in a year (January is numbered 1).
Month = nat1
inv month == month <= MONTHS_PER_YEAR;

-- A day in a month.
Day = nat1
inv day == day <= MAX_DAYS_PER_MONTH;

-- A date is a triple (year/month/day). Day of month must be consistent with respect to year.
Date :: year :Year
        month:Month
        day  :Day
inv mk_Date(y,m,d) == d <= daysInMonth(y,m);
```

The token `--` indicates a comment. Any subsequent text on the same line is ignored.

Year, month and day are each specified as an appropriate subset of natural numbers (`nat`, non-negative integers); those entities that do not allow 0 have type `nat1` (non-zero natural numbers). Note how invariants are attached to the type, specifying constraints that values of the type must always satisfy. In the case of `Year` the value must fall between `FIRST_YEAR` and `LAST_YEAR` (discussed in section 4.3), and in the case of `Month` and `Day` the value must not exceed `MONTHS_PER_YEAR` and `MAX_DAYS_PER_MONTH` respectively.

More complex is the definition of `Date`, a triple consisting of a year, a month and a day. The invariant states that the day value must be no more than the number of days in the month, with respect to the year; `daysInMonth` is a function defined later in the specification (see section 4.4). This is a good example of how one element of a data type can depend on another element (a so-called dependent type).

The general form of an invariant is a pattern separated from a truth valued expression by `==`. The pattern (which may simply be an identifier, as in `Year`) has the form of an element of the type; the expression is a constraint that all elements of the type must satisfy (and may include the free variables in the pattern, as in `Date`).

Even in this fairly simple case the value of an expressive specification language is evident; the constraints on a type are highlighted by being associated directly with the type. Only valid dates can be constructed; dates such as 29/02/2015 are excluded explicitly.

A more traditional specification would define the hierarchical data that models information of interest, then add the relevant constraints to a list of business rules, presented separately from the model and expressed in an informal manner.

To denote a date in VDM with this definition, we require an expression of the form

```
mk_Date(2016,2,29)
```

where each of the individual values satisfies its constraint (in particular, 29 is a valid day of February in year 2016). The values correspond, respectively, to the three fields `year`, `month` and `day`, and must be listed in the same order as they appear in the definition. The expression

```
mk_Date(2015,2,29)
```

on the other hand, while syntactically correct, does not satisfy the invariant. The environment for simulation and testing of a specification automatically checks all invariants to ensure the integrity of the model is maintained.

The field names can be used to reference the value in an element of the type. If we have

```
leap_day = mk_Date(2016,2,29)
```

then

```
leap_day.year  = 2016
leap_day.month = 2
leap_day.day   = 29
```

Time is defined similarly, though is a little more complex. The finest granularity is milliseconds and there is an optional time zone offset.

```
-- An hour in a day.
Hour = nat
inv hour == hour < HOURS_PER_DAY;

-- A minute in an hour.
Minute = nat
inv minute == minute < MINUTES_PER_HOUR;

-- A second in a minute.
Second = nat
inv second == second < SECONDS_PER_MINUTE;

-- A millisecond in a second.
Millisecond = nat
inv milli == milli < MILLIS_PER_SECOND;

-- A time consists of four elements (hours/minutes/seconds/milliseconds),
-- optionally with a time zone offset.
Time :: hour  :Hour
        minute:Minute
        second:Second
        milli :Millisecond
        offset:[Offset];

-- The timezone offset
Offset :: delta:Duration
          pm   :[PlusOrMinus]
          -- Offset must be less than one day and an integral number of minutes.
inv os == durLess(os.delta, ONE_DAY) and durModMinutes(os.delta) = NO_DURATION;

PlusOrMinus = <PLUS> | <MINUS>;
```

That the offset is optional n a time is indicated by brackets delimiting the type name - `offset:[Offset]`. If the offset is absent, the time is in UTC. When present the offset specifies the direction (+/-) and difference with respect to UTC. The difference is expressed as a duration that must be less than a day in length and have no finer granularity than minutes.

The definition of `PlusOrMinus` demonstrates VDM union types, that in this case is equivalent to an enumeration type in programming language. The values, bounded by angle brackets, are elements of the VDM Quote type (constants that can be compared for equality but are otherwise indivisible). However, VDM allows the union of arbitrary types so is far more general than simple enumerations.

A time is in UTC if it does not have an offset.

```
-- UTC time: a time with no offset.
UTC = Time
inv utc == utc.offset = nil;
```

The `offset` field is optional, and the absence of an optional field is expressed in VDM with the constant **`nil`**.

A combined date and time, referred to as a Date/Time Group (DTG), is defined as:

```
-- A DTG (date/time group) is a combined date and time.
DTG :: date:Date
       time:Time
inv mk_DTG(date,time) ==
      let utcTimeDur = durFromUTCTime(toUTC(time))
      in -- Adjusted time must not be earlier than 0000-01-01T00:00:00Z.
         (date = FIRST_DATE and time.offset <> nil and time.offset.pm = <PLUS> =>
              durGeq(utcTimeDur,time.offset.delta)) and
         -- Adjusted time must not be later than 9999-12-31T23:59:59,999Z
         (date = LAST_DATE and time.offset <> nil and time.offset.pm = <MINUS> =>
              durLess(durAdd(utcTimeDur,time.offset.delta),ONE_DAY));
```

The model has a minimum and maximum representable date. A combination of date and time with specific offset could result in a DTG outside the allowed bounds. When adjusted for the offset, the following values

```
0000-00-00T00:01:00+01:00
9999-12-31T23:59:59-01:00
```

are outside the allowed range of values. In the former case the offset is '+' so the time zone is one hour ahead of UTC; if we subtract the offset (one hour) from the time (one minute after the 'start of time'), the result is not valid. The `DTG` invariant ensures any value that can be constructed is in the UTC range

```
0000-00-00T00:00:00,000Z/9999-12-31T23:59:59,999Z
```

This again demonstrates the value of invariants. There are many functions that create and manipulate DTG values. By creating the invariant, and presenting it along with the data type to which it belongs, then:

- The constraint on the type is immediate and obvious;
- The constraint need only be stated once throughout the model;
- Constraint checking guarantees invalid values are identified immediately any attempt is made to construct, without needing to attach external 'validation code'.

An interval is an ordered pair of DTGs:

```
-- An interval is a pair of DTGs representing all time points between those values (inclusive).
-- The end of the interval must not be earlier than the start.
Interval :: begins:DTG
            ends  :DTG
inv ival == dtgLeq(ival.begins, ival.ends);
```

The invariant captures that the end of the interval must not be earlier than the start.

A duration is a period of time specified as a number of milliseconds:

```
-- Duration: a period of time in milliseconds.
```

```
Duration :: dur:nat;
```

The definition follows the same pattern as that of other types. In this case a duration is a number of milliseconds, so `mk_Duration(2000)` denotes 2 seconds. However, there is a difference in the the way the types are exported in the module header:

```
struct Date
Duration
```

The keyword `struct` prior to `Date` indicates the fields of `Date` – `year`, `month`, `day` – and its structure are visible outside the module. The lack of a `struct` qualifier associated with `Duration` means this is not true of `Duration`. The expression `mk_Duration(2000)` is invalid outside the module. We do not want clients to need to be aware the value is in milliseconds, which could easily be overlooked. Instead, we provide a collection of functions that create durations (from seconds, minutes, hours, etc.) and that manipulate duration values.

4.3 Values

The values section of the specification defines constants that would commonly be used in specifications and computations. Simple examples are:

```
MINUTES_PER_HOUR:nat = 60;
HOURS_PER_DAY:nat = 24;
FIRST_YEAR:nat = 0;
LAST_YEAR:nat = 9999;
```

More interesting is:

```
DAYS_PER_MONTH:map Month to nat1 = {1|->31, 2|->28, 3|->31, 4|->30, 5|->31, 6|->30,
                                    7|->31, 8|->31, 9|->30, 10|->31, 11|->30, 12|->31};
```

which demonstrates (finite) maps in VDM; it associates a month with its number of days for a non-leap year. The expression `1|->31` means the domain value 1 maps to the range value 31 (January has 31 days). The expression `DAYS_PER_MONTH(4)` is then 30, the number of days in April. For a leap year we have:

```
DAYS_PER_MONTH_LEAP:map Month to nat1 = DAYS_PER_MONTH ++ {2|->29};
```

The entire map does not need to be repeated. This says take the map for a non-leap year but 2 (February) is associated with 29 instead of 28, all other months retain the same value. Maps are a powerful abstraction in VDM; they are ideal for representing a state where the domain is typically a key that uniquely identifies the corresponding state value, and it maps to that value. (Example: the key is an identifier of a flight, and it maps to the full details of that flight.)

The range (`rng`) of a map is the set of all values that are mapped to. Since the range of `DAYS_PER_MONTH` is the set of all numbers of days in a month, then the maximum of that set is the maximum number of days in any month.

```
MAX_DAYS_PER_MONTH:nat1 = Set`max(rng DAYS_PER_MONTH);
```

``Set`max`` is a reference to the `max` function in module `Set` (see appendix E).

4.4 Functions

This section selects some of the functions defined in the module and provides commentary.

The first three functions:

- Create a UTC time (a time value without offset);
- Determine if an arbitrary time is a UTC time;
- Determine if a year is a leap year.

```
-- Create a UTC time (without milliseconds).
mkUTC: Hour * Minute * Second +> UTC
mkUTC(h,m,s) == mk_Time(h,m,s,0,nil);

-- Is a time in UTC?
isUTC: Time +> bool
isUTC(time) == time.offset = nil;

-- Is a year a leap year?
isLeap: Year +> bool
isLeap(year) == year rem 4 = 0 and (year rem 100 = 0 => year rem 400 = 0);
```

The general definition pattern is demonstrated. The first line names and declares the type of a function; for example, `isLeap` is a function that takes a year as argument and returns a truth value. The subsequent lines specify the function.

The symbol `+>` indicates the function is total: that is, for any value in the domain of the function that satisfies any pre-conditions, the function is guaranteed to terminate with a value in the range.

The function `daysInMonth` specifies the number of days in a month with respect to a year. Recall it is used to define the invariant of type `Date`.

```
-- The number of days in a month with respect to a year.
daysInMonth: Year * Month +> nat1
daysInMonth(year,month) ==
    if isLeap(year) then DAYS_PER_MONTH_LEAP(month) else DAYS_PER_MONTH(month);
```

The values section of the specification already provides the number of days in each month (section 4.3), so the function is specified in terms of those maps.

The number of days in a year is the sum of the days in the months of that year.

```
-- The number of days in a year.
daysInYear: Year +> nat1
daysInYear(year) == Seq`sum([daysInMonth(year,m) | m in set {1,...,MONTHS_PER_YEAR}]);
```

`Seq`sum` is a reference to the `sum` function in module `Seq` (see appendix D).

We want to be able to compare the various data type values that are defined in the modules, hence we must define functions that represent the order relation. In the case of `Date` we have:

```
-- Order relation on dates.
dateLess: Date * Date +> bool
dateLess(mk_Date(y1,m1,d1), mk_Date(y2,m2,d2)) ==
  y1<y2 or (y1=y2 and m1<m2) or (y1=y2 and m1=m2 and d1<d2);

-- Less than or equal relation on dates.
dateLeq: Date * Date +> bool
dateLeq(date1,date2) == dateLess(date1, date2) or date1 = date2;

-- Greater than relation on dates.
dateGrtr: Date * Date +> bool
dateGrtr(d1, d2) == dateLess(d2, d1);

-- Greater than or equal relation on dates.
dateGeq: Date * Date +> bool
dateGeq(d1, d2) == dateLeq(d2, d1);
```

The primary work is performed in `dateLess` which implements the expected order relation. The other functions are defined in terms of equality and the less than relation.

There are similar order relations on times, DTGs and durations. There is an added complication with times (and consequently DTGs). Time zone offset means two non-identical times may be equal; for example, 20:18:04Z is the same time instant as 21:18:04+01:00 (the time zone is one hour ahead of UTC). The function `normaliseTime` (described later) converts an arbitrary time to its UTC equivalent (with no time zone offset) facilitating the definition of equality and order relations over time:

```
-- Equality relation on times.
-- Primitive equality insufficient since offset must be considered.
timeEq: Time * Time +> bool
```

```
timeEq(time1, time2) == normaliseTime(time1) = normaliseTime(time2);

-- Order relation on times.
timeLess: Time * Time +> bool
timeLess(time1, time2) ==
  utcLess(normaliseTime(time1).#1, normaliseTime(time2).#1);

-- Order relation on UTC times.
utcLess: UTC * UTC +> bool
utcLess(mk_Time(h1,m1,s1,l1,-), mk_Time(h2,m2,s2,l2,-)) ==
  h1<h2 or (h1=h2 and m1<m2) or (h1=h2 and m1=m2 and s1<s2) or
  (h1=h2 and m1=m2 and s1=s2 and l1<l2);
```

Function `timeEq` normalises the arguments then tests for primitive (structural) equality. Function `timeLess` normalises the arguments then calls a subsidiary function `utcLess` that compares UTC times.

The function `overlap` defines when two intervals overlap. The definition is simple though not immediately obvious:

(begins$_1$,ends$_1$) overlaps (begins$_2$,ends$_2$) if:

$$\text{begins}_2 \le \text{ends}_1 \wedge \text{begins}_1 \le \text{ends}_2$$

```
-- Do two periods overlap?
overlap: Period * Period +> bool
overlap(p1, p2) == dtgLeq(p2.begins,p1.ends) and dtgLeq(p1.begins,p2.ends);
--post RESULT = exists d:DTG & inPeriod(d, p1) and inPeriod(d, p2);
```

Note the commented out post-condition which states the constraint more directly: two intervals overlap if there is an instant in time common to both. Type `DTG` has many elements, though it is finite. VDM interpreters cannot in general decide if a given type is finite or not. Consequently, types such as `DTG` are effectively treated as infinite, and the constraint is undecidable.

`within` defines when one interval falls wholly within another:

(begins$_1$,ends$_1$) within (begins$_2$,ends$_2$) if:

$$\text{begins}_2 \le \text{begins}_1 \wedge \text{ends}_1 \le \text{ends}_2$$

```
-- Does one interval fall wholly within another interval?
within: Interval * Interval +> bool
within(i1, i2) == dtgLeq(i2.begins,i1.begins) and dtgLeq(i1.ends,i2.ends);
--post RESULT = forall d:DTG & inInterval(d, i1) => inInterval(d, i2);
```

The post condition states any instant in the first interval is also in the second interval, and again is commented out due to quantification over a type that cannot be confirmed automatically as finite.

Many of the remaining functions are concerned with the relationship between DTG values and durations; that relationship is one of the most fundamental parts of the specification. Dates are based on the (proleptic) Gregorian calendar. Any date or DTG can be expressed as a duration commencing at time 00:00:00.000 on 1st January year 0. Many of the functions are specified in terms of the duration from 'the start of time'.

The next two definitions capture a common function: add (subtract) a duration to (from) a DTG.

```
-- Increase a DTG by a duration.
add: DTG * Duration +> DTG
add(dtg, dur) == durToDTG(durAdd(durFromDTG(dtg),dur))
post subtract(RESULT,dur) = dtg;

-- Decrease a DTG by a duration.
subtract: DTG * Duration +> DTG
subtract(dtg, dur) == durToDTG(durDiff(durFromDTG(dtg),dur))
pre durLeq(dur, durFromDTG(dtg));
--post add(RESULT,dur) = dtg;
```

In both cases the supplied DTG is converted to a duration, to (from) which the supplied duration is added (subtracted), and the result converted back to a DTG.

The post-condition of the add function states it is an inverse of the subtract function. The reserved VDM identifier **RESULT** denotes the value returned by the function, and may be referred to in the post-condition (but not the pre-condition). The equivalent post-condition is commented out in subtract. Running a simulation of the specification would cause an infinite loop if both post-conditions were made explicit: verifying the post-condition of add involves calling subtract, but verifying the post-condition of subtract involves calling add, but verifying the post-condition of add …

Note the pre-condition of subtract: it is only possible to subtract a duration from a DTG if the duration corresponding to that DTG is at least as big as the duration to be subtracted.

Function diff computes the duration between two DTGs.

```
-- The duration between two DTGs.
diff: DTG * DTG +> Duration
diff(dtg1, dtg2) == durDiff(durFromDTG(dtg1), durFromDTG(dtg2))
post (dtgLeq(dtg1,dtg2) => add(dtg1,RESULT) = dtg2) and
     (dtgLeq(dtg2,dtg1) => add(dtg2,RESULT) = dtg1);
```

The duration between two DTGs is expressed in terms of the difference between their corresponding durations. The post-condition states adding the resulting duration to the smaller of the given DTGs equals the larger of the given DTGs.

durToMinutes computes the number of whole minutes in a duration, while durFromMinutes computes the duration of a given number of minutes.

```
-- The whole number of minutes in a duration.
durToMinutes: Duration +> nat
durToMinutes(d) == durToSeconds(d) div SECONDS_PER_MINUTE
post durLeq(durFromMinutes(RESULT), d) and durLess(d, durFromMinutes(RESULT+1));

-- The duration of a number of minutes.
durFromMinutes: nat +> Duration
durFromMinutes(mn) == durFromSeconds(mn*SECONDS_PER_MINUTE);
--post durToMinutes(RESULT) = mn;
```

A duration need not be an exact number of minutes, so the post condition of durToMinutes states the duration of the resulting minutes must be no greater than the supplied duration, and the supplied duration must be less than the duration of one greater than the resulting minutes (the remainder when dividing the duration by one minute is a duration that is less than a minute).

In contrast, the post condition of durFromMinutes states if the resulting duration is converted to minutes we get the originally supplied minutes (though it is commented out to avoid infinite computation). The two functions are related, but not inverses.

Function durFromYear computes the duration of a year:

```
-- The duration of a year.
durFromYear: Year +> Duration
durFromYear(year) == durFromDays(daysInYear(year));
```

It builds on other functions by first determining the number of days in the year, then computing the duration of that many days.

The following two functions convert between a DTG and a duration.

```
-- The DTG corresponding to a duration.
durToDTG: Duration +> DTG
durToDTG(dur) == let dy = durFromDays(durToDays(dur))
                 in mk_DTG(durToDate(dy),durToTime(durDiff(dur,dy)))
post isUTC(RESULT.time) and durFromDTG(RESULT) = dur;

-- The duration of a DTG (with respect to the start of time).
durFromDTG: DTG +> Duration
durFromDTG(dtg) == let ndtg = normalise(dtg)
                   in durAdd(durFromDate(ndtg.date),durFromTime(ndtg.time));
--post durToDTG(RESULT) = dtg;
```

durToDTG and durFromDTG are inverses which is captured in the post-conditions (for the same reason as before the post-condition of durFromDTG is commented out).

Function minDTG determines the smallest DTG in a set. It demonstrates the use of quantification.

```
-- The minimum DTG in a set.
minDTG: set of DTG +> DTG
minDTG(dtgs) == iota dtg in set dtgs & forall d in set dtgs & dtgLeq(dtg, d)
pre dtgs <> {};
```

The pre-condition states the function only accepts non-empty sets (there can be no minimum element in an empty set). The function states: return the unique element in the set that is at least as small as any other element in the set. The **iota** quantifier is satisfied if the subsequent statement is true for exactly one item (in this case, the unique smallest element). The **forall** quantifier is satisfied if the subsequent statement is true for all items (in this case, the identified element is less than or equal to every element).

Quantifiers are a good way to capture specifications at an abstract level as they allow general statements to be made over potentially large sets. Quantifiers can also range over types, but if the type is infinite (or cannot be confirmed finite) the quantified statement is not decidable.

A collection of functions format elements from the model as strings per ISO 8601. Two of those functions are

```
-- Format a date and time as per ISO 8601.
format: DTG +> seq of char
format(dtg) == formatDate(dtg.date) ^ "T" ^ formatTime(dtg.time);

-- Format a date as per ISO 8601.
formatDate: Date +> seq of char
formatDate(mk_Date(y,m,d)) ==
  Numeric`zeroPad(y,4) ^ "-" ^ Numeric`zeroPad(m,2) ^ "-" ^ Numeric`zeroPad(d,2);
```

A formatted DTG is the string consisting of the formatted date followed by the separator ‘T’ followed by the formatted time. A formatted date consists of the formatted (zero padded) year, month and days values, connected by the separator ‘-’.

A DTG is normalised (transformed to UTC) by normalising its time, and if required adding (subtracting) one day to (from) the date.

```
-- Normalise a DTG value such that it is expressed as UTC; the offset is nil.
-- Applying the offset may result in a change of date.
-- Example: 2001-01-01T01:00+02:00 becomes 2000-12-31T23:00Z.
normalise: DTG +> DTG
normalise(dtg) == let mk_(ntime,pm) = normaliseTime(dtg.time),
                      baseDtg = mk_DTG(dtg.date,ntime)
                  in cases pm:
                       nil     -> baseDtg,
                       <PLUS>  -> subtract(baseDtg,ONE_DAY),
                       <MINUS> -> add(baseDtg,ONE_DAY)
                     end;
```

The time is normalised, and combined with the date. There are three possibilities:

1. Application of the offset does not change the date;
2. Application of the offset changes the date and the time zone is ahead of UTC – the date is adjusted backwards by a day;
3. Application of the offset changes the date and the time zone is behind UTC – the date is adjusted forwards by a day.

The normaliseTime function applies an offset to a time to give the corresponding UTC time. The calculated time, if necessary, wraps across a day boundary. The result indicates the direction of wrapping.

```
-- Normalise a time value to UTC with respect to the offset, wrapping across the day boundary.
-- Return an indication if the normalisation pushes the time to a different day.
-- Example: 01:00+02:00 (01:00, two hours ahead of UTC) becomes (23:00Z,<PLUS>) indicating
-- the original time with offset is on the day after the UTC time.
-- Similarly, 23:30-01:15 becomes (00:45,<MINUS>).
normaliseTime: Time +> UTC * [PlusOrMinus]
normaliseTime(time) ==
  let utcTimeDur = durFromUTCTime(tuUTC(time))
  in cases time.offset:
       nil
         -> -- Time already UTC
            mk_(time,nil),
       mk_Offset(offset,<PLUS>)
         -> -- Zone offset ahead of UTC
            if durLeq(offset,utcTimeDur)
            then -- No day change
                 mk_(durToTime(durSubtract(utcTimeDur,offset)), nil)
            else -- UTC time one day earlier
                 mk_(durToTime(durSubtract(durAdd(utcTimeDur,ONE_DAY),offset)),<PLUS>),
       mk_Offset(offset,<MINUS>)
         -> -- Zone offset behind UTC
            let adjusted = durAdd(utcTimeDur,offset)
            in if durLess(adjusted,ONE_DAY)
               then -- No day change
                    mk_(durToTime(adjusted),nil)
               else -- UTC time one day later
                    mk_(durToTime(durSubtract(adjusted,ONE_DAY)),<MINUS>)
     end;
```

This function demonstrates the use of pairs in VDM (a special instance of an n-tuple, where n≥2). The function returns an element of type `UTC*[PlusOrMinus]`; a pair whose first element is of type `UTC` and second element is of type `[PlusOrMinus]`. Given elements `utc` and `pm` of those respective types, the expression

```
mk_(utc,pm)
```

denotes the pair element. The #n operator is used to extract the individual elements from the pair, so

```
mk_(utc,pm).#1 = utc
mk_(utc,pm).#2 = pm
```

VDM supports tuples of arbitrary length. The definition of function `normalise` demonstrates the use of tuples when pattern matching.

4.5 The Set and Seq Modules

The primary purpose of this paper is to provide a commentary on the date/time specification. However, that specification employs other low level specifications for manipulating sets, sequences, numeric values, and character values. In this section we provide some commentary on the Set and Seq modules.

In these cases, there are no type definitions. We are simply specifying general purpose functions over sets and sequences to supplement the primitive functions provided by VDM.

The cross product of two sets is the set of all pairs whose first element is drawn from one set and second element from the other.

```
-- The cross product of two sets.
xProduct[@a,@b]: set of @a * set of @b +> set of (@a * @b)
xProduct(s,t) == { mk_(x,y) | x in set s, y in set t }
post card RESULT = card s * card t;
```

The specification demonstrates the use of comprehensions which in some situations can be used to capture the meaning in a concise manner. In this case it says create the set of all pairs whose first element is drawn from `s` and second element is drawn from `t` (also referred to as the Cartesian product).

The post condition states the number of items in the resulting set is the product of the number of items in the argument sets.

The specification also demonstrates polymorphism in VDM. The function takes as arguments a set of one kind of thing (denoted by @a), and a set of another kind of thing (denoted by @b), and delivers a set of pairs. The first element of each pair is a @a and the second element of each pair is a @b. The cross product does not care what types @a and @b actually denote, so the function is polymorphic.

The function xform in the Seq module takes a sequence and a function, and delivers the result of applying the function to each element in the sequence, i.e. it transforms the sequence. This is generally called the 'map' function but we avoid that name here due to the primitive VDM type of the same name.

```
-- Apply a function to all elements of a sequence.
xform[@a,@b]: (@a+>@b) * seq of @a +> seq of @b
xform(f,s) == [ f(s(i)) | i in set inds s ]
post len RESULT = len s;
```

Like xProduct, xform is a polymorphic function with two type arguments; all we care is that the function transforms a value of one type to another type, not what the actual types are.

The elements in a sequence are indexed commencing at 1, so s(i) is the ith element of sequence s. The expression **inds** s denotes the set of all indexes of s, so if s has *n* elements, then **inds** s is the set {1,2,…,n}.

The definition of xform further demonstrates the value of comprehensions, capturing directly that the function argument f is being applied to each item in the sequence s. The pattern "i **in set inds** s" is the standard way in which comprehensions range over all items in a sequence; that is, we range over the set of indices of the sequence.

The function fold1 in the Seq module takes a non-empty sequence and a binary function, and delivers the result of iterating the function over the elements in the sequence.

```
-- Fold (iterate, accumulate, reduce) a binary function over a non-empty sequence.
-- The function is assumed to be associative.
fold1[@a]: (@a * @a +> @a) * seq1 of @a +> @a
fold1(f, s) == cases s:
                 [e]   -> e,
                 s1^s2 -> f(fold1[@a](f,s1), fold1[@a](f,s2))
               end
--pre forall x,y,z:@a & f(x,f(y,z)) = f(f(x,y),z)
measure size1;

size1[@a]: @a * seq of @a +> nat
size1(-, s) == len s;
```

For example, folding the addition operator over a numeric sequence, such as

```
fold1[nat](add, [1,2,3,4])
```

is equivalent to

```
add(1,add(2,add(3,4)))
```

or

```
1 + 2 + 3 + 4
```

The function add is defined in the Numeric module (appendix C). When a polymorphic function (fold1) is called, the instance of the type parameter (nat) must be stated explicitly between the function name and its arguments.

The specification demonstrates pattern matching on sequences in VDM. Case analysis is performed on the argument `s` for the the singleton sequence, and a sequence of two or more items[3]. Most interesting is the latter case, where the pattern is `s1^s2`. This says the matched sequence (`s`) is split into two sub-sequences which, when concatenated, give the original sequence. The only guarantee is the sub-sequences contain at least one item each. The specification cannot make any assumptions about how the sequence is sub-divided when matching. If it was required, for example, to specifically break it into the first item and the remainder, the pattern would be `[e]^s2`.

The pre-condition states the function argument must be associative. Since the function is polymorphic, any type could be instantiated, including infinite types. Consequently, the pre-condition is commented out to allow simulation of the model.

Finally, note the specification of a **`measure`** function. `fold1` is a recursive function, thus during simulation may lead to infinite computation. A measure function takes the same arguments as the function being specified, and returns a natural number (non-negative integer) as result. The value of the measure function should be smaller in the recursive call than in the main function. If the measure function reduces on each call, and the result can never be less than 0, then the function is guaranteed to terminate. This is another condition that can be checked during simulation. If the measure function is not less on the recursive call, the simulation highlights an error in the specification.

5 Definitions

API	Application Programming Interface
DTG	Date-Time Group; a date in the Gregorian calendar and time with optional offset from UTC.
IETF	Internet Engineering Task Force
ISO	International Organization for Standardization
RFC	Request For Comments
UTC	Co-ordinated Universal Time
VDM	Vienna Definition Method
VDM-SL	VDM Specification Language

6 References

[1] ISO 8601:2004 Representation of dates and times, at http://www.iso.org/iso/catalogue_detail?csnumber=40874

[2] IETF RFC 3339 Date and Time on the Internet: Timestamps, at https://www.ietf.org/rfc/rfc3339.txt

[3] Vienna Development Method, at https://en.wikipedia.org/wiki/Vienna_Development_Method

[4] VDM-10 Language Manual, TR-001, version 7

[5] Modelling Systems – Practical Tools and Techniques in Software Development, Second Edition, J. Fitzgerald and P. G. Larsen, Cambridge University Press, 2009

[3] The sequence argument of `fold1` has type `seq1 of @a`, the type of non-empty sequences, so the function would fail if invoked with the empty sequence. Consequently, the case analysis does not need to account for the empty sequence.

[6] Combinatorial Testing for VDM, in: Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods, 2010, pp. 278-285. P. G. Larsen, K. Lausdahl and N. Battle.

[7] Overture Tool, at http://overturetool.org/

A. ISO8601 Module

```
/*
   A model of dates, times, intervals and durations. Intended as a core library for use by
   higher level models that require dates and/or times and/or intervals and/or durations.

   The model is based on the ISO 8601 standard for representation of dates and times using
   the Gregorian calendar:

     https://en.wikipedia.org/wiki/ISO_8601

   For dates and times, this model largely conforms to the RFC 3339 profile:

     https://tools.ietf.org/html/rfc3339

   Exceptions to RFC 3339 are:
   - A seconds value of 60 (leap second) is not supported;
   - ISO 8601/RFC 3339 impose no limitation on the resolution of times; the finest granularity
     of this model is milliseconds.

   The model additionally supports a subset of the specification of time intervals and
   durations from ISO 8601. Those aspects supported are:
   - a duration is expressed as a number of milliseconds;
   - an interval is expressed as a start/end date/time value.

   All functions are explicit and executable. Where a non-executable condition adds value, it
   is included as a comment.

*/
module ISO8601
imports from Numeric all,
        from Set all,
        from Seq all
exports types struct Year
              struct Month
              struct Day
              struct Hour
              struct Minute
              struct Second
              struct Millisecond
              struct Date
              struct Time
              struct Offset
              struct UTC
              struct DTG
              struct Interval
              Duration
        values MILLIS_PER_SECOND, SECONDS_PER_MINUTE, MINUTES_PER_HOUR, HOURS_PER_DAY,
FIRST_YEAR, LAST_YEAR: nat
               DAYS_PER_MONTH, DAYS_PER_MONTH_LEAP: map nat1 to nat1
               MAX_DAYS_PER_MONTH, MONTHS_PER_YEAR, DAYS_PER_YEAR, DAYS_PER_LEAP_YEAR: nat1
               FIRST_DATE, LAST_DATE: Date;
               NO_DURATION, ONE_MILLISECOND, ONE_SECOND, ONE_MINUTE, ONE_HOUR, ONE_DAY,
ONE_YEAR, ONE_LEAP_YEAR: Duration
        functions mkUTC: Hour * Minute * Second +> UTC
                  isUTC: Time +> bool
                  toUTC: Time +> UTC
                  isLeap: Year +> bool
                  daysInMonth: Year * Month +> nat1
                  daysInYear: Year +> nat1
                  dateLess: Date * Date +> bool
                  dateLeq: Date * Date +> bool
                  dateGrtr: Date * Date +> bool
                  dateGeq: Date * Date +> bool
                  timeEq: Time * Time +> bool
                  timeLess: Time * Time +> bool
                  utcLess: UTC * UTC +> bool
                  timeLeq: Time * Time +> bool
                  timeGrtr: Time * Time +> bool
                  timeGeq: Time * Time +> bool
                  dtgEq: DTG * DTG +> bool
                  dtgLess: DTG * DTG +> bool
                  dtgLeq: DTG * DTG +> bool
                  dtgGrtr: DTG * DTG +> bool
                  dtgGeq: DTG * DTG +> bool
                  durLess: Duration * Duration +> bool
                  durLeq: Duration * Duration +> bool
                  durGrtr: Duration * Duration +> bool
                  durGeq: Duration * Duration +> bool
                  dtgInRange: DTG * DTG * DTG +> bool
                  inInterval: DTG * Interval +> bool
                  overlap: Interval * Interval +> bool
                  within: Interval * Interval +> bool
                  add: DTG * Duration +> DTG
                  subtract: DTG * Duration +> DTG
                  diff: DTG * DTG +> Duration
                  durAdd: Duration * Duration +> Duration
                  durSubtract: Duration * Duration +> Duration
                  durMultiply: Duration * nat +> Duration
                  durDivide: Duration * nat +> Duration
                  durDiff: Duration * Duration +> Duration
                  durToMillis: Duration +> nat
                  durFromMillis: nat +> Duration
                  durToSeconds: Duration +> nat
                  durFromSeconds: nat +> Duration
                  durToMinutes: Duration +> nat
                  durFromMinutes: nat +> Duration
                  durModMinutes : Duration +> Duration
                  durToHours: Duration +> nat
                  durFromHours: nat +> Duration
                  durModHours : Duration +> Duration
                  durToDays: Duration +> nat
                  durFromDays : nat +> Duration
                  durModDays : Duration +> Duration
                  durToMonth: Duration * Year +> nat
                  durFromMonth: Year * Month +> Duration
                  durUptoMonth: Year * Month +> Duration
                  durToYear : Duration * Year +> nat
                  durFromYear: Year +> Duration
                  durUptoYear: Year +> Duration
                  durToDTG: Duration +> DTG
                  durFromDTG: DTG +> Duration
                  durToDate: Duration +> Date
                  durFromDate: Date +> Duration
                  durToTime: Duration +> UTC
                  durFromTime: Time +> Duration
                  durFromInterval: Interval +> Duration
                  minDTG: set of DTG +> DTG
                  maxDTG: set of DTG +> DTG
                  minDate: set of Date +> Date
                  maxDate: set of Date +> Date
                  minTime: set of Time +> Time
                  maxTime: set of Time +> Time
                  minDuration: set of Duration +> Duration
                  maxDuration: set of Duration +> Duration
                  sumDuration: seq of Duration +> Duration
                  instant: DTG +> Interval
                  format: DTG +> seq of char
                  formatDate: Date +> seq of char
                  formatTime: Time +> seq of char
                  formatInterval: Interval +> seq of char
                  formatDuration: Duration +> seq of char
                  normalise: DTG +> DTG

definitions

types

  -- A year: 0 = 0AD (or 1BC).
  Year = nat
  inv year == FIRST_YEAR <= year and year <= LAST_YEAR;

  -- A month in a year (January is numbered 1).
  Month = nat1
  inv month == month <= MONTHS_PER_YEAR;

  -- A day in a month.
  Day = nat1
  inv day == day <= MAX_DAYS_PER_MONTH;

  -- An hour in a day.
  Hour = nat
  inv hour == hour < HOURS_PER_DAY;

  -- A minute in an hour.
  Minute = nat
  inv minute == minute < MINUTES_PER_HOUR;

  -- A second in a minute.
  Second = nat
  inv second == second < SECONDS_PER_MINUTE;

  -- A millisecond in a second.
  Millisecond = nat
  inv milli == milli < MILLIS_PER_SECOND;

  -- A date is a triple (year/month/day).
  -- Day of month must be consistent with respect to year.
  Date :: year :Year
          month:Month
          day  :Day
  inv mk_Date(y,m,d) == d <= daysInMonth(y,m);

  -- A time consists of four elements (hours/minutes/seconds/milliseconds),
  -- optionally with a time zone offset.
  Time :: hour  :Hour
          minute:Minute
          second:Second
          milli :Millisecond
          offset:[Offset];

  -- The timezone offset
  Offset :: delta:Duration
            pm   :[PlusOrMinus]
            -- Offset must be less than one day and an integral number of minutes.
  inv os == durLess(os.delta, ONE_DAY) and durModMinutes(os.delta) = NO_DURATION;

  PlusOrMinus = <PLUS> | <MINUS>;

  -- UTC time: a time with no offset.
  UTC = Time
  inv utc == utc.offset = nil;

  -- A DTG (date/time group) is a combined date and time.
  DTG :: date:Date
         time:Time
  inv mk_DTG(date,time) ==
        let utcTimeDur = durFromUTCTime(toUTC(time))
        in -- Adjusted time must not be earlier than 0000-01-01T00:00:00Z.
           (date = FIRST_DATE and time.offset <> nil and time.offset.pm = <PLUS> =>
                durGeq(utcTimeDur,time.offset.delta)) and
           -- Adjusted time must not be later than 9999-12-31T23:59:59,999Z
           (date = LAST_DATE and time.offset <> nil and time.offset.pm = <MINUS> =>
                durLess(durAdd(utcTimeDur,time.offset.delta),ONE_DAY));

  -- An interval is a pair of DTGs representing all time instants between those
  -- bounding values (inclusive).
  -- The end of the interval must not be earlier than the start.
  Interval :: begins:DTG
              ends  :DTG
  inv ival == dtgLeq(ival.begins, ival.ends);

  -- Duration: a period of time in milliseconds.
  Duration :: dur:nat;

values

  MILLIS_PER_SECOND:nat = 1000;
  SECONDS_PER_MINUTE:nat = 60;
  MINUTES_PER_HOUR:nat = 60;
  HOURS_PER_DAY:nat = 24;
  DAYS_PER_MONTH:map nat1 to nat1 = {1|->31, 2|->28, 3|->31, 4|->30, 5|->31, 6|->30,
                                     7|->31, 8|->31, 9|->30, 10|->31, 11|->30, 12|->31};
  DAYS_PER_MONTH_LEAP:map nat1 to nat1 = DAYS_PER_MONTH ++ {2|->29};
  MAX_DAYS_PER_MONTH:nat1 = Set`max(rng DAYS_PER_MONTH);
  MONTHS_PER_YEAR:nat1 = card dom DAYS_PER_MONTH;
  DAYS_PER_YEAR:nat1 = daysInYear(1); -- 1 is an arbitrary non-leap year.
  DAYS_PER_LEAP_YEAR:nat1 = daysInYear(4); -- 4 is an arbitrary leap year.
  FIRST_YEAR:nat = 0;
  LAST_YEAR:nat = 9999;
  FIRST_DATE:Date = mk_Date(FIRST_YEAR,1,1);
  LAST_DATE:Date = mk_Date(LAST_YEAR,12,31);
  NO_DURATION:Duration = durFromMillis(0);
  ONE_MILLISECOND:Duration = durFromMillis(1);
  ONE_SECOND:Duration = durFromSeconds(1);
  ONE_MINUTE:Duration = durFromMinutes(1);
  ONE_HOUR:Duration = durFromHours(1);
  ONE_DAY:Duration = durFromDays(1);
  ONE_YEAR:Duration = durFromDays(DAYS_PER_YEAR);
  ONE_LEAP_YEAR:Duration = durFromDays(DAYS_PER_LEAP_YEAR);

functions

  -- Create a UTC time (without milliseconds).
  mkUTC: Hour * Minute * Second +> UTC
  mkUTC(h,m,s) == mk_Time(h,m,s,0,nil);

  -- Is a time in UTC?
  isUTC: Time +> bool
  isUTC(time) == time.offset = nil;

  -- Drop the offset part of a time.
  toUTC: Time +> UTC
  toUTC(time) == mu(time, offset|->nil);

  -- Is a year a leap year?
  isLeap: Year +> bool
  isLeap(year) == year rem 4 = 0 and (year rem 100 = 0 => year rem 400 = 0);

  -- The number of days in a month with respect to a year.
  daysInMonth: Year * Month +> nat1
  daysInMonth(year,month) ==
    if isLeap(year) then DAYS_PER_MONTH_LEAP(month) else DAYS_PER_MONTH(month);

  -- The number of days in a year.
  daysInYear: Year +> nat1
  daysInYear(year) == Seq`sum ([daysInMonth(year,m) | m in set {1,...,MONTHS_PER_YEAR}]);

  -- Order relation on dates.
  dateLess: Date * Date +> bool
  dateLess(mk_Date(y1,m1,d1), mk_Date(y2,m2,d2)) ==
    y1<y2 or (y1=y2 and m1<m2) or (y1=y2 and m1=m2 and d1<d2);

  -- Less than or equal relation on dates.
  dateLeq: Date * Date +> bool
  dateLeq(date1,date2) == dateLess(date1, date2) or date1 = date2;

  -- Greater than relation on dates.
  dateGrtr: Date * Date +> bool
  dateGrtr(d1, d2) == dateLess(d2, d1);

  -- Greater than or equal relation on dates.
  dateGeq: Date * Date +> bool
  dateGeq(d1, d2) == dateLeq(d2, d1);

  -- Equality relation on times.
  -- Primitive equality insufficient since offset must be considered.
  timeEq: Time * Time +> bool
  timeEq(time1, time2) == normaliseTime(time1) = normaliseTime(time2);

  -- Order relation on times.
  timeLess: Time * Time +> bool
  timeLess(time1, time2) ==
    utcLess(normaliseTime(time1).#1, normaliseTime(time2).#1);
```

```
-- Order relation on UTC times.
utcLess: UTC * UTC +> bool
utcLess(mk_Time(h1,m1,s1,l1,-), mk_Time(h2,m2,s2,l2,-)) ==
  h1<h2 or (h1=h2 and m1<m2) or (h1=h2 and m1=m2 and s1<s2) or
  (h1=h2 and m1=m2 and s1=s2 and l1<l2);

-- Less than or equal relation on times.
timeLeq: Time * Time +> bool
timeLeq(time1, time2) == timeLess(time1, time2) or timeEq(time1, time2);

-- Greater than relation on times.
timeGrtr: Time * Time +> bool
timeGrtr(d1, d2) == timeLess(d2, d1);

-- Greater than or equal relation on times.
timeGeq: Time * Time +> bool
timeGeq(d1, d2) == timeLeq(d2, d1);

-- Equality relation on DTGs: are their normalised values identical?
-- Primitive equality insufficient since primitive equality on times is insufficient.
dtgEq: DTG * DTG +> bool
dtgEq(dtg1, dtg2) == normalise(dtg1) = normalise(dtg2);

-- Order relation on DTGs.
dtgLess: DTG * DTG +> bool
dtgLess(dtg1, dtg2) ==
  let n1 = normalise(dtg1),
      n2 = normalise(dtg2)
  in dateLess(n1.date,n2.date) or (n1.date=n2.date and utcLess(n1.time,n2.time));

-- Less than or equal relation on DTGs.
dtgLeq: DTG * DTG +> bool
dtgLeq(dtg1, dtg2) == dtgLess(dtg1, dtg2) or dtgEq(dtg1, dtg2);

-- Greater than relation on DTGs.
dtgGrtr: DTG * DTG +> bool
dtgGrtr(d1, d2) == dtgLess(d2, d1);

-- Greater than or equal relation on DTGs.
dtgGeq: DTG * DTG +> bool
dtgGeq(d1, d2) == dtgLeq(d2, d1);

-- Order relation on durations.
durLess: Duration * Duration +> bool
durLess(d1, d2) == d1.dur < d2.dur;

-- Less than or equal relation on durations.
durLeq: Duration * Duration +> bool
durLeq(d1, d2) == durLess(d1,d2) or d1 = d2;

-- Greater than relation on durations.
durGrtr: Duration * Duration +> bool
durGrtr(d1, d2) == durLess(d2, d1);

-- Greater than or equal relation on durations.
durGeq: Duration * Duration +> bool
durGeq(d1, d2) == durLeq(d2, d1);

-- Does a DTG fall between two given DTGs?
dtgInRange: DTG * DTG * DTG +> bool
dtgInRange(dtg1, dtg2, dtg3) == dtgLeq(dtg1, dtg2) and dtgLeq(dtg2, dtg3);

-- Does a DTG fall within an interval?
inInterval: DTG * Interval +> bool
inInterval(dtg, ival) == dtgInRange(ival.begins, dtg, ival.ends);

-- Do two intervals overlap?
overlap: Interval * Interval +> bool
overlap(i1, i2) == dtgLeq(i2.begins,i1.ends) and dtgLeq(i1.begins,i2.ends);
--post RESULT = exists d:DTG & inInterval(d, i1) and inInterval(d, i2);

-- Does one interval fall wholly within another interval?
within: Interval * Interval +> bool
within(i1, i2) == dtgLeq(i2.begins,i1.begins) and dtgLeq(i1.ends,i2.ends);
```

```
--post RESULT = forall d:DTG & inInterval(d, i1) => inInterval(d, i2);

-- Increase a DTG by a duration.
add: DTG * Duration +> DTG
add(dtg, dur) == durToDTG(durAdd(durFromDTG(dtg),dur))
post RESULT.time.offset = dtg.time.offset and subtract(RESULT,dur) = dtg;

-- Decrease a DTG by a duration.
subtract: DTG * Duration +> DTG
subtract(dtg, dur) == durToDTG(durDiff(durFromDTG(dtg),dur))
pre durLeq(dur, durFromDTG(dtg))
post RESULT.time.offset = dtg.time.offset;
--post add(RESULT,dur) = dtg;

-- The duration between two DTGs.
diff: DTG * DTG +> Duration
diff(dtg1, dtg2) == durDiff(durFromDTG(dtg1), durFromDTG(dtg2))
post (dtgLeq(dtg1,dtg2) => add(dtg1,RESULT) = dtg2) and
     (dtgLeq(dtg2,dtg1) => add(dtg2,RESULT) = dtg1);

-- Add two durations.
durAdd: Duration * Duration +> Duration
durAdd(d1, d2) == mk_Duration(d1.dur + d2.dur)
--post durDiff(RESULT, d1) = d2 and durDiff(RESULT,d2) = d1;
post durSubtract(RESULT, d1) = d2 and
     durSubtract(RESULT, d2) = d1;

-- Subtract on duration from another.
durSubtract: Duration * Duration +> Duration
durSubtract(d1, d2) == mk_Duration(d1.dur - d2.dur)
pre durGeq(d1, d2);
--post durAdd(RESULT, d2) = d1;

-- Multiply a duration by a fixed amount.
durMultiply: Duration * nat +> Duration
durMultiply(d, n) == mk_Duration(d.dur * n)
post durDivide(RESULT, n) = d;

-- Divide a duration by a fixed amount.
durDivide: Duration * nat +> Duration
durDivide(d, n) == mk_Duration(d.dur div n);
--post durLeq(durMultiply(RESULT, n), d) and durLess(d, durMultiply(RESULT, n+1));

-- The difference between two durations.
durDiff: Duration * Duration +> Duration
durDiff(d1, d2) == mk_Duration(abs(d1.dur - d2.dur))
post (durLeq(d1,d2) => durAdd(d1,RESULT)=d2) and (durLeq(d2,d1) => durAdd(d2,RESULT)=d1);

-- The whole number of milliseconds in a duration.
durToMillis: Duration +> nat
durToMillis(d) == d.dur
post durFromMillis(RESULT) = d;

-- The duration of a number of milliseconds.
durFromMillis: nat +> Duration
durFromMillis(sc) == mk_Duration(sc);
--post durToMillis(RESULT) = sc;

-- The whole number of seconds in a duration.
durToSeconds: Duration +> nat
durToSeconds(d) == durToMillis(d) div MILLIS_PER_SECOND
post durLeq(durFromSeconds(RESULT), d) and durLess(d, durFromSeconds(RESULT+1));

-- The duration of a number of seconds.
durFromSeconds: nat +> Duration
durFromSeconds(sc) == durFromMillis(sc*MILLIS_PER_SECOND);
--post durToSeconds(RESULT) = sc;

-- The whole number of minutes in a duration.
durToMinutes: Duration +> nat
durToMinutes(d) == durToSeconds(d) div SECONDS_PER_MINUTE
post durLeq(durFromMinutes(RESULT), d) and durLess(d, durFromMinutes(RESULT+1));

-- The duration of a number of minutes.
```

```
durFromMinutes: nat +> Duration
durFromMinutes(mn) == durFromSeconds(mn*SECONDS_PER_MINUTE);
--post durToMinutes(RESULT) = mn;

-- Remove all whole minutes from a duration.
durModMinutes : Duration +> Duration
durModMinutes(d) == mk_Duration(d.dur rem ONE_MINUTE.dur)
post durLess(RESULT, ONE_MINUTE);
--exists n:nat & durAdd(durFromMinutes(n),RESULT) = d

-- The whole number of hours in a duration.
durToHours: Duration +> nat
durToHours(d) == durToMinutes(d) div MINUTES_PER_HOUR
post durLeq(durFromHours(RESULT), d) and durLess(d, durFromHours(RESULT+1));

-- The duration of a number of hours.
durFromHours: nat +> Duration
durFromHours(hr) == durFromMinutes(hr*MINUTES_PER_HOUR);
--post durToHours(RESULT) = hr;

-- Remove all whole hours from a duration.
durModHours : Duration +> Duration
durModHours(d) == mk_Duration(d.dur rem ONE_HOUR.dur)
post durLess(RESULT, ONE_HOUR);
--exists n:nat & durAdd(durFromHours(n),RESULT) = d

-- The whole number of days in a duration.
durToDays: Duration +> nat
durToDays(d) == durToHours(d) div HOURS_PER_DAY
post durLeq(durFromDays(RESULT), d) and durLess(d, durFromDays(RESULT+1));

-- The duration of a number of days.
durFromDays: nat +> Duration
durFromDays(dy) == durFromHours(dy*HOURS_PER_DAY);
--post durToDays(RESULT) = dy;

-- Remove all whole days from a duration.
durModDays : Duration +> Duration
durModDays(d) == mk_Duration(d.dur rem ONE_DAY.dur)
post durLess(RESULT, ONE_DAY);
--exists n:nat & durAdd(durFromDays(n),RESULT) = d

-- The whole number of months in a duration (with respect to a year).
durToMonth: Duration * Year +> nat
durToMonth(dur, year) ==
  Set`max({ m | m in set {1,...,MONTHS_PER_YEAR} & durLeq(durUptoMonth(year,m), dur) }) - 1
pre durLess(dur,durFromYear(year));

-- The duration of a month (with respect to a year).
durFromMonth: Year * Month +> Duration
durFromMonth(year, month) == durFromDays(daysInMonth(year,month));

-- The duration up to the start of a month (with respect to a year).
durUptoMonth: Year * Month +> Duration
durUptoMonth(year, month) == sumDuration([durFromMonth(year,m) | m in set {1,...,month-1}]);

-- The whole number of years in a duration (starting from a reference year).
durToYear : Duration * Year +> nat
durToYear(dur, year) ==
  if durLess (dur, durFromYear(year))
  then 0
  else 1 + durToYear(durDiff(dur, durFromYear(year)), year+1)
--post RESULT = Set`max({ y | y : Year & durLeq(durUptoYear(year+y), dur) })
measure durToYear_measure;

-- The measure function for durToYear
durToYear_measure : Duration * Year +> nat
durToYear_measure(d,-) == d.dur;

-- The duration of a year.
durFromYear: Year +> Duration
durFromYear(year) == durFromDays(daysInYear(year));

-- The duration up to the start of a year.
```

```
durUptoYear: Year +> Duration
durUptoYear(year) == sumDuration([durFromYear(y) | y in set {FIRST_YEAR,...,year-1}]);

-- The DTG corresponding to a duration.
durToDTG: Duration +> DTG
durToDTG(dur) == let dy = durFromDays(durToDays(dur))
                 in mk_DTG(durToDate(dy),durToTime(durDiff(dur,dy)))
post isUTC(RESULT.time) and durFromDTG(RESULT) = dur;

-- The duration of a DTG (with respect to the start of time).
durFromDTG: DTG +> Duration
durFromDTG(dtg) == let ndtg = normalise(dtg)
                   in durAdd(durFromDate(ndtg.date),durFromTime(ndtg.time));
--post durToDTG(RESULT) = dtg;

-- The date corresponding to a duration.
durToDate: Duration +> Date
durToDate(dur) == let yr = durToYear(dur,FIRST_YEAR),
                      ydur = durDiff(dur, durUptoYear(yr)),
                      mn = durToMonth(ydur,yr)+1,
                      dy = durToDays(durDiff(ydur, durUptoMonth(yr,mn)))+1
                  in mk_Date(yr,mn,dy)
post durLeq(durFromDate(RESULT), dur) and durLess(dur, durAdd(durFromDate(RESULT),ONE_DAY));

-- The duration of a date (with respect to the start of time).
durFromDate: Date +> Duration
durFromDate(date) ==
  durAdd(durUptoYear(date.year),
         durAdd(durUptoMonth(date.year,date.month), durFromDays(date.day-1)));
--post durToDate(RESULT) = date;

-- The time corresponding to a duration.
durToTime: Duration +> UTC
durToTime(dur) == let hr = durToHours(dur),
                      mn = durToMinutes(durDiff(dur,durFromHours(hr))),
                      hmd = durAdd(durFromHours(hr),durFromMinutes(mn)),
                      sc = durToSeconds(durDiff(dur,hmd)),
                      ml = durToMillis(durDiff(dur,durAdd(hmd,durFromSeconds(sc))))
                  in mk_Time(hr,mn,sc,ml,nil)
pre durLess(dur,ONE_DAY)
post durFromTime(RESULT) = dur;

-- The duration of a time.
durFromTime: Time +> Duration
durFromTime(time) ==
  let ntime = normaliseTime(time).#1
  in durFromUTCTime(ntime);
--post timeEq(durToTime(RESULT), time);

-- The duration of a UTC time; offset correction not necessary.
durFromUTCTime: UTC +> Duration
durFromUTCTime(time) ==
  durAdd(durFromHours(time.hour),
         durAdd(durFromMinutes(time.minute),
                durAdd(durFromSeconds(time.second),durFromMillis(time.milli))));
--post durToTime(RESULT) = time;

-- The duration of a time interval.
durFromInterval: Interval +> Duration
durFromInterval(i) == diff(i.begins, i.ends)
post add(i.begins, RESULT) = i.ends;

-- The minimum DTG in a set.
minDTG: set of DTG +> DTG
minDTG(dtgs) == iota dtg in set dtgs & forall d in set dtgs & dtgLeq(dtg, d)
pre dtgs <> {};

-- The maximum DTG in a set.
maxDTG: set of DTG +> DTG
maxDTG(dtgs) == iota dtg in set dtgs & forall d in set dtgs & dtgLeq(d, dtg)
pre dtgs <> {};

-- The minimum Date in a set.
minDate: set of Date +> Date
```

```
minDate(dates) == iota date in set dates & forall d in set dates & dateLeq(date, d)
pre dates <> {};

-- The maximum Date in a set.
maxDate: set of Date +> Date
maxDate(dates) == iota date in set dates & forall d in set dates & dateLeq(d, date)
pre dates <> {};

-- The minimum Time in a set.
minTime: set of Time +> Time
minTime(times) == iota time in set times & forall t in set times & timeLeq(time, t)
pre times <> {};

-- The maximum Time in a set.
maxTime: set of Time +> Time
maxTime(times) == iota time in set times & forall t in set times & timeLeq(t, time)
pre times <> {};

-- The minimum Duration in a set.
minDuration: set of Duration +> Duration
minDuration(durs) == iota dur in set durs & forall d in set durs & durLeq(dur, d)
pre durs <> {};

-- The maximum Duration in a set.
maxDuration: set of Duration +> Duration
maxDuration(durs) == iota dur in set durs & forall d in set durs & durLeq(d, dur)
pre durs <> {};

-- The sum of a sequence of durations.
sumDuration: seq of Duration +> Duration
sumDuration(sd) == mk_Duration(Seq`sum([ sd(i).dur | i in set inds sd ]));

-- An interval that represents an instant in time.
instant: DTG +> Interval
instant(dtg) == mk_Interval(dtg,dtg)
post inInterval(dtg, RESULT);
     --and forall d:DTG & dtgEq(d,dtg) <=> inInterval(d,RESULT);

-- Format a date and time as per ISO 8601.
format: DTG +> seq of char
format(dtg) == formatDate(dtg.date) ^ "T" ^ formatTime(dtg.time);

-- Format a date as per ISO 8601.
formatDate: Date +> seq of char
formatDate(mk_Date(y,m,d)) ==
  Numeric`zeroPad(y,4) ^ "-" ^ Numeric`zeroPad(m,2) ^ "-" ^ Numeric`zeroPad(d,2);

-- Format a time as per ISO 8601.
formatTime: Time +> seq of char
formatTime(mk_Time(h,m,s,l,o)) ==
  let frac = if l = 0 then "" else "," ^ Numeric`zeroPad(l,3),
      os = if o = nil then "Z" else formatOffset(o)
  in Numeric`zeroPad(h,2) ^ ":" ^ Numeric`zeroPad(m,2) ^ ":" ^
     Numeric`zeroPad(s,2) ^ frac ^ os;

-- Format a time offset as per ISO 8601.
formatOffset: Offset +> seq of char
formatOffset(mk_Offset(dur,pm)) ==
  let hm = durToTime(dur),
           sign = if pm = <PLUS> then "+" else "-"
  in sign ^ Numeric`zeroPad(hm.hour,2) ^ ":" ^ Numeric`zeroPad(hm.minute,2);

-- Format a DTG interval as per ISO 8601.
formatInterval: Interval +> seq of char
formatInterval(interval) == format(interval.begins) ^ "/" ^ format(interval.ends);

-- Format a duration as per ISO 8601.
formatDuration: Duration +> seq of char
formatDuration(d) ==
  let numDays = durToDays(d),
      mk_Time(h,m,s,l,-) = durToTime(durModDays(d)),
      item: nat * char +> seq of char
      item(n,c) == if n = 0 then "" else Numeric`formatNat(n)^[c],
      itemSec: nat * nat +> seq of char
```

```
        itemSec(x,y) == Numeric`formatNat(x) ^ "." ^ Numeric`zeroPad(y,3) ^ "S",
        date = item(numDays,'D'),
        time = item(h,'H') ^ item(m,'M') ^ if l=0 then item(s,'S') else itemSec(s,l)
    in if date="" and time=""
       then "PT0S"
       else "P" ^ date ^ (if time="" then "" else "T" ^ time);

  -- Normalise a DTG value such that it is expressed as UTC; the offset is nil.
  -- Applying the offset may result in a change of date.
  -- Example: 2001-01-01T01:00+02:00 becomes 2000-12-31T23:00Z.
  normalise: DTG +> DTG
  normalise(dtg) == let mk_(ntime,pm) = normaliseTime(dtg.time),
                        baseDtg = mk_DTG(dtg.date,ntime)
                    in cases pm:
                         nil     -> baseDtg,
                         <PLUS>  -> subtract(baseDtg,ONE_DAY),
                         <MINUS> -> add(baseDtg,ONE_DAY)
                       end;

  -- Normalise a time value to UTC with respect to the offset, wrapping across the day
  -- boundary. Return an indication if the normalisation pushes the time to a different day.
  -- Example: 01:00+02:00 (01:00, two hours ahead of UTC) becomes (23:00Z,<PLUS>) indicating
  -- the original time with offset is on the day after the UTC time.
  -- Similarly, 23:30-01:15 becomes (00:45,<MINUS>).
  normaliseTime: Time +> UTC * [PlusOrMinus]
  normaliseTime(time) ==
    let utcTimeDur = durFromUTCTime(toUTC(time))
    in cases time.offset:
         nil
           -> -- Time already UTC
              mk_(time,nil),
         mk_Offset(offset,<PLUS>)
           -> -- Zone offset ahead of UTC
              if durLeq(offset,utcTimeDur)
              then -- No day change
                   mk_(durToTime(durSubtract(utcTimeDur,offset)), nil)
              else -- UTC time one day earlier
                   mk_(durToTime(durSubtract(durAdd(utcTimeDur,ONE_DAY),offset)),<PLUS>),
         mk_Offset(offset,<MINUS>)
           -> -- Zone offset behind UTC
              let adjusted = durAdd(utcTimeDur,offset)
              in if durLess(adjusted,ONE_DAY)
                 then -- No day change
                      mk_(durToTime(adjusted),nil)
                 else -- UTC time one day later
                      mk_(durToTime(durSubtract(adjusted,ONE_DAY)),<MINUS>)
       end;

end ISO8601
```

B. Char Module

```
/*
   A module that specifies and defines general purpose types, constants and functions over
   characters and strings (sequences characters).

   All functions are explicit and executable. Where a non-executable condition adds value, it
   is included as a comment.
*/
module Char
exports types String
              String1
              Upper
              Lower
              Letter
              Digit
              Octal
              Hex
              AlphaNum
              AlphaNumUpper
              AlphaNumLower
              Space
              WhiteSpace
              Phrase
              PhraseUpper
              PhraseLower
              Text
              TextUpper
              TextLower
        values SP, TB, CR, LF, WHITE_SPACE, UPPER, LOWER, DIGIT, OCTAL, HEX: char
        functions padLeft: String * char * nat +> String
                  padRight: String * char * nat +> String

definitions

types

  String = seq of char;

  String1 = seq1 of char;

  Upper = char
  inv c == c in set UPPER;

  Lower = char
  inv c == c in set LOWER;

  Letter = Upper | Lower;

  Digit = char
  inv c == c in set DIGIT;

  Octal = char
  inv c == c in set OCTAL;

  Hex = char
  inv c == c in set HEX;

  AlphaNum = Letter | Digit;

  AlphaNumUpper = Upper | Digit;

  AlphaNumLower = Lower | Digit;

  Space = char
  inv sp == sp = ' ';

  WhiteSpace = char
  inv ws == ws in set WHITE_SPACE;

  Phrase = seq1 of (AlphaNum|Space);

  PhraseUpper = seq1 of (AlphaNumUpper|Space);

  PhraseLower = seq1 of (AlphaNumLower|Space);

  Text = seq1 of (AlphaNum|WhiteSpace);

  TextUpper = seq1 of (AlphaNumUpper|WhiteSpace);

  TextLower = seq1 of (AlphaNumLower|WhiteSpace);

values

  SP:char = ' ';
  TB:char = '\t';
  CR:char = '\r';
  LF:char = '\n';
  WHITE_SPACE:set of char = {SP,TB,CR,LF};
  UPPER:set of char = {'A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q',
                       'R','S','T','U','V','W','X','Y','Z'};
  LOWER:set of char = {'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q',
                       'r','s','t','u','v','w','x','y','z'};
  DIGIT:set of char = {'0','1','2','3','4','5','6','7','8','9'};
  OCTAL:set of char = {'0','1','2','3','4','5','6','7'};
  HEX:set of char = {'0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F'};
```

```
functions

  -- Pad a string on the left with a given character up to a specified length.
  padLeft: String * char * nat +> String
  padLeft(sq,c,n) == [ c | i in set {1 ,..., n - len sq} ] ^ sq;

  -- Pad a string on the right with a given character up to a specified length.
  padRight: String * char * nat +> String
  padRight(sq,c,n) == sq ^ [ c | i in set {1 ,..., n - len sq} ];

end Char
```

C. Numeric Module

```
/*
   A module that specifies and defines general purpose functions over numerics.

   All definitions are explicit and executable.
*/
module Numeric
imports from Char all
exports functions min: real * real +> real
                  max: real * real +> real
                  formatNat: nat +> seq of Char`Digit
                  zeroPad: nat * nat1 +> seq of Char`Digit
                  formatNat: nat +> seq of Char`Digit
                  fromChar: Char`Digit +> nat
                  toChar: nat +> Char`Digit
                  add: real * real +> real
                  mult: real * real +> real

definitions

functions

  -- The minimum of two numerics.
  min: real * real +> real
  min(x,y) == if x<y then x else y;

  -- The maximum of two numerics.
  max: real * real +> real
  max(x,y) == if x>y then x else y;

  -- Format a natural number as a string of digits.
  formatNat: nat +> seq of Char`Digit
  formatNat(n) == if n < 10
                  then [toChar(n)]
                  else formatNat(n div 10) ^ [toChar(n mod 10)]
  measure size1;

  -- Convert a character digit to the corresponding natural number.
  fromChar: Char`Digit +> nat
  fromChar(c) == cases c:
                  '0' -> 0,
                  '1' -> 1,
                  '2' -> 2,
                  '3' -> 3,
                  '4' -> 4,
                  '5' -> 5,
                  '6' -> 6,
                  '7' -> 7,
                  '8' -> 8,
                  '9' -> 9
                 end
  post toChar(RESULT) = c;

  -- Convert a numeric digit to the corresponding character.
  toChar: nat +> Char`Digit
  toChar(n) == cases n:
                 0 -> '0',
                 1 -> '1',
                 2 -> '2',
                 3 -> '3',
                 4 -> '4',
                 5 -> '5',
                 6 -> '6',
                 7 -> '7',
                 8 -> '8',
                 9 -> '9'
               end
  pre 0 <= n and n <= 9;
  --post fromChar(RESULT) = n

  -- Format a natural number as a string with leading zeros up to a specified length.
  zeroPad: nat * nat1 +> seq of Char`Digit
  zeroPad(n,w) == Char`padLeft(formatNat(n),'0',w);

  -- The following functions wrap primitives for convenience, to allow them for example to
  -- serve as function arguments.

  -- Sum of two numbers.
  add: real * real +> real
  add(m,n) == m+n;

  -- Product of two numbers.
  mult: real * real +> real
  mult(m,n) == m*n;

  -- Measure functions.

  size1: nat +> nat
  size1(n) == n;

end Numeric
```

D. Seq Module

```
/*
   A module that specifies and defines general purpose functions over sequences.

   All functions are explicit and executable. Where a non-executable condition adds value, it
   is included as a comment.
*/
module Seq
imports from Numeric all
exports functions sum: seq of real +> real
                  prod: seq of real +> real
                  min: seq1 of real +> real
                  max: seq1 of real +> real
                  inSeq[@a]: @a * seq of @a +> bool
                  numOccurs[@a]: @a * seq of @a +> nat
                  permutation[@a]: seq of @a * seq of @a +> bool
                  preSeq[@a]: seq of @a * seq of @a +> bool
                  postSeq[@a]: seq of @a * seq of @a +> bool
                  subSeq[@a]: seq of @a * seq of @a +> bool
                  xform[@a,@b]: (@a +> @b) * seq of @a +> seq of @b
                  fold[@a]: (@a * @a +> @a) * @a * seq of @a +> @a
                  fold1[@a]: (@a * @a +> @a) * seq1 of @a +> @a
                  zip[@a,@b]: seq of @a * seq of @b +> seq of (@a * @b)
                  unzip[@a,@b]: seq of (@a * @b) +> seq of @a * seq of @b
                  isDistinct[@a]: seq of @a +> bool
                  app[@a]: seq of @a * seq of @a +> seq of @a
                  setOf[@a]: seq of @a +> set of @a

definitions

functions

  -- The sum of a sequence of numerics.
  sum: seq of real +> real
  sum(s) == fold[real](Numeric`add,0,s);

  -- The product of a sequence of numerics.
```

```
prod: seq of real +> real
prod(s) == fold[real](Numeric`mult,1,s);

-- The minimum of a sequence of numerics.
min: seq1 of real +> real
min(s) == fold1[real](Numeric`min,s)
post RESULT in set elems s and forall e in set elems s & RESULT <= e;

-- The maximum of a sequence of numerics.
max: seq1 of real +> real
max(s) == fold1[real](Numeric`max,s)
post RESULT in set elems s and forall e in set elems s & RESULT >= e;

-- Does an element appear in a sequence?
inSeq[@a]: @a * seq of @a +> bool
inSeq(e,s) == e in set elems s;

-- The number of times an element appears in a sequence.
numOccurs[@a]: @a * seq of @a +> nat
numOccurs(e,sq) == len [ 0 | i in set inds sq & sq(i) = e ];

-- Is one sequence a permutation of another?
permutation[@a]: seq of @a * seq of @a +> bool
permutation(sq1,sq2) ==
  len sq1 = len sq2 and
  forall i in set inds sq1 & numOccurs[@a](sq1(i),sq1) = numOccurs[@a](sq1(i),sq2);

-- Is one sequence a prefix of another?
preSeq[@a]: seq of @a * seq of @a +> bool
preSeq(pres,full) == pres = full(1,...,len pres);

-- Is one sequence a suffix of another?
postSeq[@a]: seq of @a * seq of @a +> bool
postSeq(posts,full) == preSeq[@a](reverse posts, reverse full);

-- Is one sequence a subsequence of another sequence?
subSeq[@a]: seq of @a * seq of @a +> bool
subSeq(sub,full) == exists i,j in set inds full & sub = full(i,...,j);

-- Apply a function to all elements of a sequence.
xform[@a,@b]: (@a+>@b) * seq of @a +> seq of @b
xform(f,s) == [ f(s(i)) | i in set inds s ]
post len RESULT = len s;

-- Fold (iterate, accumulate, reduce) a binary function over a sequence.
-- The function is assumed to be associative and have an identity element.
fold[@a]: (@a * @a +> @a) * @a * seq of @a +> @a
fold(f, e, s) == cases s:
                   []    -> e,
                   [x]   -> x,
                   s1^s2 -> f(fold[@a](f,e,s1), fold[@a](f,e,s2))
                 end
--pre (exists x:@a & forall y:@a & f(x,y) = y and f(y,x) = y)
--and forall x,y,z:@a & f(x,f(y,z)) = f(f(x,y),z)
measure size2;

-- Fold (iterate, accumulate, reduce) a binary function over a non-empty sequence.
-- The function is assumed to be associative.
fold1[@a]: (@a * @a +> @a) * seq1 of @a +> @a
fold1(f, s) == cases s:
                 [e]   -> e,
                 s1^s2 -> f(fold1[@a](f,s1), fold1[@a](f,s2))
               end
--pre forall x,y,z:@a & f(x,f(y,z)) = f(f(x,y),z)
measure size1;

-- Pair the corresponding elements of two lists of equal length.
zip[@a,@b]: seq of @a * seq of @b +> seq of (@a * @b)
zip(s,t) == [ mk_(s(i),t(i)) | i in set inds s ]
pre len s = len t
post len RESULT = len s;

-- Split a list of pairs into a list of firsts and a list of seconds.
unzip[@a,@b]: seq of (@a * @b) +> seq of @a * seq of @b
```

```
  unzip(s) == mk_([ s(i).#1 | i in set inds s], [ s(i).#2 | i in set inds s])
  post let mk_(t,u) = RESULT in len t = len s and len u = len s;

  -- Are the elements of a list distinct (no duplicates).
  isDistinct[@a]: seq of @a +> bool
  isDistinct(s) == len s = card elems s;

  -- The following functions wrap primitives for convenience, to allow them for example to
  -- serve as function arguments.

  -- Concatenation of two sequences.
  app[@a]: seq of @a * seq of @a +> seq of @a
  app(m,n) == m^n;

  -- Set of sequence elements.
  setOf[@a]: seq of @a +> set of @a
  setOf(s) == elems(s);

  -- Measure functions.

  size1[@a]: (@a * @a +> @a) * seq1 of @a +> nat
  size1(-, s) == len s;

  size2[@a]: (@a * @a +> @a) * @a * seq of @a +> nat
  size2(-, -, s) == len s;

end Seq
```

E. Set Module

```
/*
   A module that specifies and defines general purpose functions over sets.

   All functions are explicit and executable. Where a non-executable condition adds value, it
   is included as a comment.
*/
module Set
imports from Numeric all,
        from Seq all
exports functions sum: set of real +> real
                  prod: set of real +> real
                  min: set of real +> real
                  max: set of real +> real
                  toSeq[@a]: set of @a +> seq of @a
                  xform[@a,@b]: (@a +> @b) * set of @a +> set of @b
                  fold[@a]: (@a * @a +> @a) * @a * set of @a +> @a
                  fold1[@a]: (@a * @a +> @a) * set of @a +> @a
                  pairwiseDisjoint[@a]: set of set of @a +> bool
                  isPartition[@a]: set of set of @a * set of @a +> bool
                  permutations[@a]: set of @a +> set of seq1 of @a
                  xProduct[@a,@b]: set of @a * set of @b +> set of (@a * @b)

definitions

functions

  -- The sum of a set of numerics.
  sum: set of real +> real
  sum(s) == fold[real](Numeric`add,0,s);

  -- The product of a set of numerics.
  prod: set of real +> real
  prod(s) == fold[real](Numeric`mult,1,s);

  -- The minimum of a set of numerics.
  min: set of real +> real
  min(s) == fold1[real](Numeric`min, s)
  pre s <> {}
  post RESULT in set s and forall e in set s & RESULT <= e;

  -- The maximum of a set of numerics.
  max: set of real +> real
```

```
max(s) == fold1[real](Numeric`max, s)
pre s <> {}
post RESULT in set s and forall e in set s & RESULT >= e;

-- The sequence whose elements are those of a specified set, with no duplicates.
-- No order is guaranteed in the resulting sequence.
toSeq[@a]: set of @a +> seq of @a
toSeq(s) == fold[@a](Seq`app,[],s)
post len RESULT = card s and forall e in set s & Seq`inSeq[@a](e,RESULT);

-- Apply a function to all elements of a set. The result set may be smaller than the
-- argument set if the function argument is not injective.
xform[@a,@b]: (@a+>@b) * set of @a +> set of @b
xform(f,s) == { f(e) | e in set s }
post (forall e in set s & f(e) in set RESULT) and
     (forall r in set RESULT & exists e in set s & f(e) = r);

-- Fold (iterate, accumulate, reduce) a binary function over a set.
-- The function is assumed to be commutative and associative, and have an identity element.
fold[@a]: (@a * @a +> @a) * @a * set of @a +> @a
fold(f, e, s) == cases s:
                   {}        -> e,
                   {x}       -> x,
                   t union u -> f(fold[@a](f,e,t), fold[@a](f,e,u))
                 end
--pre (exists x:@a & forall y:@a & f(x,y) = y and f(y,x) = y)
--and (forall x,y:@a & f(x, y) = f(y, x))
--and (forall x,y,z:@a & f(x,f(y,z)) = f(f(x,y),z))
measure size2;

-- Fold (iterate, accumulate, reduce) a binary function over a non-empty set.
-- The function is assumed to be commutative and associative.
fold1[@a]: (@a * @a +> @a) * set of @a +> @a
fold1(f, s) == cases s:
                 {e}       -> e,
                 t union u -> f(fold1[@a](f,t), fold1[@a](f,u))
               end
pre s <> {}
--and (forall x,y:@a & f(x,y) = f(y,x))
--and (forall x,y,z:@a & f(x,f(y,z)) = f(f(x,y),z))
measure size1;

-- Are the members of a set of sets pairwise disjoint.
pairwiseDisjoint[@a]: set of set of @a +> bool
pairwiseDisjoint(ss) == forall x,y in set ss & x<>y => x inter y = {};

-- Is a set of sets a partition of a set?
isPartition[@a]: set of set of @a * set of @a +> bool
isPartition(ss,s) == pairwiseDisjoint[@a](ss) and dunion ss = s;

-- All (sequence) permutations of a set.
permutations[@a]: set of @a +> set of seq1 of @a
permutations(s) ==
  cases s:
    {e} -> {[e]},
    -   -> dunion { { [e]^tail | tail in set permutations[@a](s\{e}) } | e in set s }
  end
pre s <> {}
post -- for a set of size n, there are n! permutations
     card RESULT = prod({1,...,card s}) and
     forall sq in set RESULT & len sq = card s and elems sq = s
measure size;

-- The cross product of two sets.
xProduct[@a,@b]: set of @a * set of @b +> set of (@a * @b)
xProduct(s,t) == { mk_(x,y) | x in set s, y in set t }
post card RESULT = card s * card t;

-- Measure functions.

size[@a]: set of @a +> nat
size(s) == card s;

size1[@a]: (@a * @a +> @a) * set of @a +> nat
```

```
    size1(-, s) == card s;

    size2[@a]: (@a * @a +> @a) * @a * set of @a +> nat
    size2(-, -, s) == card s;

  end Set
```