The OCaml documentation for FV Time

Formal Vindications S.L.*

September, 2022

UNIVERSITAT DE
BARCELONA

Unió Europea
Fons Europeu
de Desenvolupament Regional

*Project funded by the Spanish Ministry of Science, Innovation and Universities, the State Agency for Research and the European Regional Development Fund (ERDF).

Contents

1 Context

The FV Time Library is a Coq-verified implementation of the UTC standard with some extra utilities to make it more usable for both critical and regular programming. Currently, this library is only available in the OCaml language[1].

In this manual we present the documentation for the extract OCaml code, to use natively from other OCaml applications.

2 File structure

The files directly extracted from Coq to OCaml, and thus verified, are `FVTM.ml` (the code itself) and `FVTM.mli` (the headers). They can be used directly as OCaml libraries. They make use of the Coq implementation for OCaml primitive integers. Below we describe the files and what each of them does.

- `FVTM.ml`, `FVTM.mli`: The code and headers extracted directly from Coq.
- `uint63.ml`, `uint63.mli`: The code and headers provided by Coq for the extraction to OCaml's primitive machine integers.

3 Compilation

Any method for compilation of OCaml using modules works. If you have an OCaml development, FV Time's code files just need to be added to the it to compile, as shown in Figure 1.

4 The UTC standard

The UTC (Coordinated Universal Time) is the most widespread standard for measuring and representing points in time. The counting of time is done according to *atomic clocks*.

UTC uses the Gregorian calendar to count days, which is the usual calendar: it accounts for the year, the month and the day within the month. Some years are leap and have an extra day in February according to the rule:

- Every year divisible by 400 is a leap year.
- Every year divisible by 4 but not by 100 is also a leap year.
- No other year is a leap year.

Seconds are counted following an atomic clock, i.e. all seconds have the exact same duration. But, since the Earth's rotation period slightly varies due to physical interactions, not all solar days have a duration of 86400 SI seconds. If no adjustment was made, the solar time would gradually differ from UTC. To avoid this effect, the UTC standard introduces **leap seconds**.

Leap seconds are introduced by a committee of experts (IERS) with at least six months of advance, and are impredictable in the long term. The only way to account for them is keeping an updated list of the past leap seconds. Theoretically, leap seconds can be positive (if added) or negative (if removed), but since the Earth tends to slow down, no negative leap second has ever happened.

The standard says that, when it occurs, a positive leap second is inserted between second 23:59:59 of a chosen UTC calendar date and second 00:00:00 of the following date. The definition of UTC states that the last day of December and June are preferred, with the last day of March or September as second preference, and the last day of any other month as third preference. All leap seconds (as of 2017) have been scheduled for either June 30 or December 31. The extra second is displayed on UTC clocks as 23:59:60. A negative

[1] As of now, we only extract to OCaml because it is the only language that has plans to have a verified extraction process.

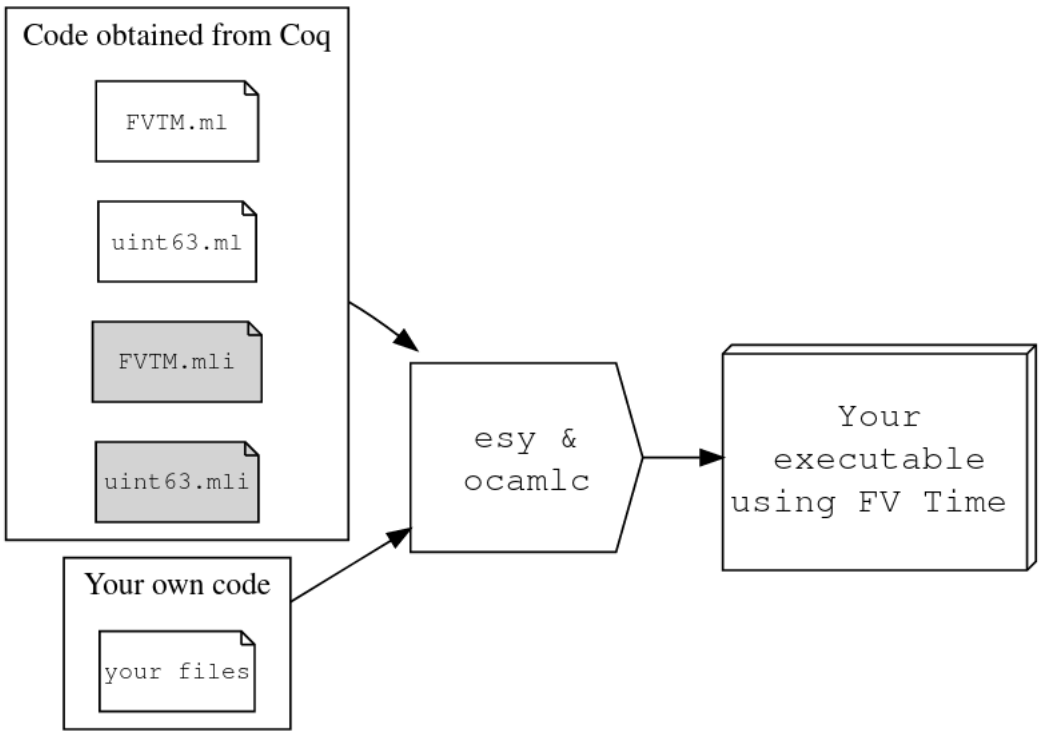

Figure 1: Diagram for the compilation process.

leap second would suppress second 23:59:59 of the last day of a chosen month, so that second 23:59:58 of that date would be followed immediately by second 00:00:00 of the following date.

Although the UTC standard includes the format `"Y-M-D h:m:s"`, time by atomic clocks is actually counted as a number of seconds (units of time) elapsed since some chosen point of time called epoch. This number of seconds is called **timestamp**. The most common epoch used, and the one our library chooses, is 1970-01-01 00:00:00. Then by definition its timestamp is 0 and any other timestamp starts counting from that time on.

UTC differs from International Atomic Time (TAI) by 37, the number of leap seconds that have been added. Most computer systems claim to implement UTC or GMT (which is not an official name) but actually implement TAI, or Unix, which is equal to UTC not counting leap seconds.

5 A new notion for consistent time arithmetic

Practically all commercial libraries for dealing with times and timestamps use Unix, even if they claim to use UTC. When adding and subtracting durations or time intervals to a given time, an issue arises due to the irregular periods that the Gregorian calendar and UTC define. For systems that work in Unix, the issue arises with months and years, because they don't have a constant duration. What these libraries do is to define an artificial operation on months and years that doesn't respect basic arithmetical properties.

They define *adding a month* as adding one to the month component of the time. But of course, the result of this operation is not always correct. For example, adding a month in this sense to `2009-01-31 14:32:54` yields `2009-02-31 14:32:54`, which is not a valid time because February doesn't have 31 days. Then, the adopted solution is correct the wrong component by going back to the previous valid one, so the result would be `2009-02-28 14:32:54`. Similarly, they can add any number to the month component, carrying to the year if necessary, and then correct the wrong component. For example, adding 24 months to `2008-02-29 15:00:00` gives `2010-02-28 15:00:00`. Analogously, the operation for adding years is defined.

This operation does not behave as addition does. For example, if in the first example we subtract one month to the result, we don't get the original time. Instead, we get `2009-01-28 14:32:54`. We can say that $1 - 1 \neq 0$!

Our library uses UTC, which means that this problem affects all the components to the left of seconds. Not all minutes have the same duration, nor all hours, nor all days. Our solution is to implement two different types of operations in time arithmetic. The first one, that we call shift functions, follows the same logic that other libraries follow with months and years, but with all the components. The second one is a definition of a new standard for durations called formal time, and operations called add-formal with it that behave consistently with basic arithmetical properties.

5.1 Shift functions

In that context, we extrapolate the logic described above for months and years of libraries working in Unix to the rest of the components for the UTC case. The shift function *shifts* a component of the time, carrying to the left if necessary, and then if the result is invalid performs corrections on the wrong component(s) to give a specific close previous valid time.

An example would be *subtracting two days* to the time `2016-12-31 23:59:60` (a leap second). With the procedure above before the correction, the result is `2016-12-29 23:59:60`. Since on that date there was no leap second, a correction must be performed on the wrong component to give the previous valid time, hence the final result is `2016-12-29 23:59:59`.

We chose the shift name because these functions are not proper addition. We call them `shift_utc` functions, and they have six instances: `shift_utc_seconds`, `shift_utc_minutes`, `shift_utc_hours`, `shift_utc_days`, `shift_utc_months`, and `shift_utc_years`.

5.2 Add-formal functions

In order to have time arithmetic with the usual arithmetical properties, we have define a new standard called formal time, that establishes standard durations for every component. A formal second is an atomic second, and a formal minute is 60 formal seconds, etc. See `formalTime` for all the details.

We have chosen to define formal months as 30 formal days. Therefore, *adding a formal month* is a clear arithmetical operation, consisting in adding a *constant* number of seconds (30 * 24 * 60 * 60 seconds). In the example above, adding a formal month to `2009-01-31 14:32:54` yields `2009-03-02 14:32:54`. The result is always valid by construction (except when it goes beyond the minimum or maximum date).

Now, *subtracting two formal days* to the time `2016-12-31 23:59:60` yields `2016-12-30 00:00:00`, which lets us see that the constant 86400 *cdot* 2 has been subtracted.

We define these functions with the name `add_formal`, and they have six instances: `add_formal_seconds`, `add_formal_minutes`, `add_formal_hours`, `add_formal_days`, `add_formal_months`, and `add_formal_years`. We also have the generic `add_formal` and `subtract_formal`.

These functions satisfy desirable arithmetical properties, for example if $t_1 + \Delta t = t_2$, then $t_2 - \Delta t = t_1$, i.e., $1 - 1 = 0$.

Notice that the only add-formal function that behaves exactly as its shift counterpart is `add_formal_seconds`, which is exactly the same as `shift_utc_seconds`.

6 FV Time Library: Main features

1. Fully **formally verified** using the Coq proof assistant.
2. **Fully in UTC**, including **leap seconds**.
3. Definition of the **formal time standard**: Durations are not well-defined in UTC, because due to leap seconds minutes, hours, etc. do not have constant durations. Our library solves two critical issues by introducing the `formalTime` standard for the time durations in UTC:
   - Give constant definitions for durations or time intervals, allowing to group durations in units bigger than seconds consistently.
   - Regarding **time arithmetic**, the library implements two kind of functions for addition and subtraction of `time` and durations.
     - The `shift_utc` functions work as is common in other libraries (the detailed description is below). Since durations are not constant, they do not satisfy certain basic arithmetical properties as the one described in the next item.
     - The `add_formal` functions are a new definition of addition and subtraction that works with the `formalTime` standard, hence satisfy for a time `t` and integer `n`:
       [Property of `add_formal` functions:] `add_formal_X (add_formal_X t n) -n = t`.
       In an improper non-functional language, this means that if $t_1 + \Delta t = t_2$, then $t_2 - \Delta t = t_1$.
4. Includes a function to check the version number and last updated date. Different versions may give different results due to the addition of leap seconds.

7 Module `FVTM` : The OCaml extraction of FV Time

`type int63 = Uint63.t`

A machine 63-bit integer. It can be used as signed or unsigned, depending on the function. Each function controls the range of its input.

Unsigned integers range from 0 to 9223372036854775807.

Signed integers range from -4611686018427387904 to 4611686018427387903.

```
type date =
{  year : int63 ;
  month : int63 ;
  day : int63 ;
}
```

A record for dates. Valid values range from 1970-1-1 to 9999-12-31 and need to be valid UTC dates in the sense of `FVTM.valid_date`[➦].

```
type time =
{  date_of_time : date ;
  hour : int63 ;
  minute : int63 ;
  second : int63 ;
}
```

A record for times. Valid values range from 1970-1-1 00:00:00 to 9999-12-31 23:59:59 and need to be valid UTC times in the sense of `FVTM.valid_time`[➦].

```
type clock =
{  chour : int63 ;
  cminute : int63 ;
  csecond : int63 ;
}
```

A record for clocks, i.e., the hour-minute-second part of a time. Valid values need to satisfy:

- $0 \leq$ `chour` ≤ 23
- $0 \leq$ `cminute` ≤ 59
- $0 \leq$ `csecond` ≤ 60 (because we account for leap seconds)

```
type formalTime =
{  fyears : int63 ;
  fmonths : int63 ;
  fdays : int63 ;
  fhours : int63 ;
  fminutes : int63 ;
  fseconds : int63 ;
}
```

A record for `formalTime`, composed of six values that represent a time interval duration in formal time units:

- 1 formal second = 1 atomic second
- 1 formal minute = 60 formal seconds
- 1 formal hour = 60 formal minutes
- 1 formal day = 24 formal hours
- 1 formal month = 30 formal days
- 1 formal year = 365 formal days

The value of each component is restricted to avoid overflowing the maximum `FVTM.int63`[➦] when operating with a `formalTime`. The following must hold:

- $0 \leq$ `fyears` ≤ 2924712086,
- $0 \leq$ `fmonths` ≤ 35583997055,
- $0 \leq$ `fdays` ≤ 1067519911673,
- $0 \leq$ `fhours` ≤ 25620477880152,
- $0 \leq$ `fminutes` ≤ 1537228672809129,
- $0 \leq$ `fseconds` ≤ 92233720368547757.

These are the only restrictions for a `formalTime` when given as an argument. As output, a `formalTime` is always given in *normal form*, meaning that the components satisfy the following conditions too:

- `fseconds` < 60,
- `fminutes` < 60,
- `fhours` < 24,
- `fdays` < 30,
- `fmonths` ≤ 12, and if `fdays` ≤ 5 then `fmonths` < 12.

This last condition is because 12 formal months are 360 formal days, less than a formal year. Thus, it is allowed to say 0-12-4-0-0-0, but if we want to add one more formal day, then the expression becomes 1-0-0-0-0-0.

```
type 'a possibly =
  | Result of 'a
```

There are no errors and the result of the operation is returned

```
  | InvalidDate of date
```

The given `FVTM.date`[➦] is invalid. Only dates in UTC (starting in 1970, ending in 9999) are accepted

```
  | InvalidTime of time
```

The given `FVTM.time`[➦] is invalid. Only times in UTC (with leap seconds and starting in 1970, ending in 9999) are accepted

```
  | InvalidFormalTime of formalTime
```

The given `FVTM.formalTime`[➦] is invalid. Either some component is negative, or it is too big and would cause overflow

```
  | InvalidYear of int63
```

The given year is invalid. Only years between 1970 and 9999 are accepted

```
  | InvalidMonth of int63
```

The given month is invalid. Months are a number between 1 and 12

```
  | InvalidDay of int63 * int63 * int63
```

The given day is invalid. Only days between 1 and `FVTM.days_of_month`[➦] of the given year and month are accepted

```
  | InvalidHour of int63
```

The given hour is invalid. Hours are a number between 0 and 23

```
  | InvalidMinute of int63
```

The given minute is invalid. Minutes are a number between 0 and 59

```
  | InvalidSecond of date * int63 * int63 * int63
```

The given second is invalid. Only seconds between 0 and `FVTM.max_second`[➦] of the given date, hour and minute are accepted

| `InvalidClockSecond of int63`

The given clock second is invalid. Clock seconds are a number between 0 and 60

| `NumberOutOfBounds of int63`

The given integer is out of bounds. Operating with it would lead to overflowing the minimum or maximum time

| `Overflow`

Overflow: with the input you gave, the resulting time would be before 1970 or after 9999

| `InvalidOrderOfTimes of time * time`

The first argument is smaller than the second one. Time difference can only be computed if the first argument is greater than the second one

A record for the several errors that Coq can detect.

`val max_year : int63`

`val is_leap_year_plain : int63 -> bool`

A function which takes a natural number and, interpreting it as a year in the Gregorian calendar, returns `true` if it is a leap year and `false` otherwise.

`val is_leap_year : int63 -> bool possibly`

If the given year is between 1970 and 9999, then returns a `Result` that satisfies the specification of `FVTM.is_leap_year_plain`[➦].

Errors:

- `InvalidYear` if the given year `year` does **not** satisfy $1970 \leq$ `year` ≤ 9999

`val days_of_month_plain : int63 -> int63 -> int63`

A function which takes two natural numbers and, interpreting the first one as a year in the Gregorian calendar and the second one as a month, returns the number of days that the input month has on the input year.

`val days_of_month : int63 -> int63 -> int63 possibly`

If the given year is between 1970 and 9999 and the given month is between 1 and 12, then `FVTM.days_of_month`[➦] returns a `Result` that satisfies the specification of `FVTM.days_of_month_plain`[➦].

Errors:

- `InvalidYear` if the first argument `year` does **not** satisfy $1970 \leq$ `year` ≤ 9999
- `InvalidMonth` if the second argument `month` does **not** satisfy $1 \leq$ `month` ≤ 12

`val valid_date : date -> bool`

Checks for the validity of a date in UTC. Given a date Y-M-D, valid or invalid, returns `true` if and only if all the following hold:

- $1970 \leq \mathrm{Y} \leq 9999$;
- $1 \leq \mathrm{M} \leq 12$;
- $1 \leq \mathrm{D} \leq$ `FVTM.days_of_month`[➦] Y M.

`val mkDate : int63 -> int63 -> int63 -> date possibly`

Produces a `Result` `FVTM.date`[➦] if the given arguments `year`, `month` and `day` are valid.

Errors:

- `InvalidYear` if **not** $1970 \leq$ `year` ≤ 9999
- `InvalidMonth` if **not** $1 \leq$ `month` ≤ 12
- `InvalidDay` if **not** $1 \leq$ `day` $\leq$ `FVTM.days_of_month`[➦] `year month`

`type leapSeconds = (date * bool) list`

A type for expressing the list of leap seconds. Each element of the list represents a leap second, via a pair with the `FVTM.date`[➦] in which the leap second happened, and a `bool` expressing if it was positive (`false`) or negative (`true`).

`val version_number : (Uint63.t * Uint63.t) * Uint63.t`

Returns the version number of FVTM, relevant for the leap seconds

`val version_date : date`

Returns the date in which this version of FVTM was released, relevant for leap seconds.

`val ls2022 : leapSeconds`

The list of leap seconds as of May 2022

`val le_date_plain : date -> date -> bool`

A function which takes two dates as input, and returns `true` if the first one is smaller or equal to the second one and `false` otherwise, according to the chronological order.

`val lt_date_plain : date -> date -> bool`

A function which takes two dates as input, and returns `true` if the first one is strictly smaller than the second one and `false` otherwise, according to the chronological order.

`val le_date : date -> date -> bool possibly`

If the given dates are valid, returns a `Result` that satisfies the specification of `FVTM.le_date_plain`[➦].

Errors:

- `InvalidDate` if any of the arguments is not a `FVTM.valid_date`[➦]

`val lt_date : date -> date -> bool possibly`

If the given dates are valid, returns a `Result` that satisfies the specification of `FVTM.lt_date_plain`[➦].

Errors:

- `InvalidDate` if any of the arguments is not a `FVTM.valid_date`[➦]

`val max_second_plain : leapSeconds -> date -> int63 -> int63 -> int63`

A function which takes a list of leap seconds, a date and two natural numbers representing the hour and minute, and returns the number of seconds that the minute had on that date and hour according to the list of leap seconds.

`val max_second : date -> int63 -> int63 -> int63 possibly`

If the given date is valid, the hour is at most 23 and the minute is at most 59, returns a `Result` that satisfies the specification of `FVTM.max_second_plain`[➦] applied with the leap seconds of `FVTM.ls2022`[➦].

Errors:

- `InvalidDate` if for the given date `d`, `FVTM.valid_date`[➦] `d` = `false`
- `InvalidHour` if for the given hour `h`, **not** $0 \leq$ `h` ≤ 23
- `InvalidMinute` if for the given minute `m`, **not** $0 \leq$ `m` ≤ 59

`val valid_time : time -> bool`

Checks for the validity of a time in UTC. Given a time `Y-M-D h:m:s`, valid or invalid, returns `true` if and only if all the following hold:

- `FVTM.valid_date`[➦] `Y-M-D` = `true`;
- $0 \leq$ `h` ≤ 23;
- $0 \leq$ `m` ≤ 59;
- $0 \leq$ `s` $\leq$ `FVTM.max_second_plain`[➦] `Y-M-D h m`.

`val clock_of_time_plain : time -> clock`

A function which takes a `FVTM.time`[➦] and returns its corresponding `FVTM.clock`[➦].

`val clock_of_time : time -> clock possibly`

If the given time is valid, returns a `Result` that satisfies the specification of `FVTM.clock_of_time_plain`[➦].

Errors:

- `InvalidTime` if for the given time `t`, `FVTM.valid_time`[➦] `t` = `false`

`val mkClock : int63 -> int63 -> int63 -> clock possibly`

Produces a `Result FVTM.clock`[➦] if the given arguments `chour`, `cminute` and `csecond` are valid.

Errors:

- `InvalidHour` if **not** $0 \leq$ `hour` ≤ 23
- `InvalidMinute` if **not** $0 \leq$ `minute` ≤ 59
- `InvalidClockSecond` if **not** $0 \leq$ `second` ≤ 60

```
val mkTime :
  int63 ->
  int63 ->
  int63 ->
  int63 -> int63 -> int63 -> time possibly
```

Produces a `Result FVTM.time`[➦] if the given arguments `year`, `month`, `day`, `hour`, `minute` and `second` are valid.

Errors:

- `InvalidYear` if **not** $1970 \leq$ `year` ≤ 9999

- `InvalidMonth` if **not** $1 \leq$ `month` ≤ 12
- `InvalidDay` if **not** $1 \leq$ `day` $\leq$ `FVTM.days_of_month`[➦] `year month`
- `InvalidHour` if **not** $0 \leq$ `hour` ≤ 23
- `InvalidMinute` if **not** $0 \leq$ `minute` ≤ 59
- `InvalidSecond` if **not** $0 \leq$ `second` $\leq$ `FVTM.max_second`[➦] `year-month-day hour minute`

`val le_time_plain : time -> time -> bool`

A function which takes two times as input, and returns `true` if the first one is smaller or equal to the second one and `false` otherwise, according to the chronological order.

`val lt_time_plain : time -> time -> bool`

A function which takes two times as input, and returns `true` if the first one is strictly smaller than the second one and `false` otherwise, according to the chronological order.

`val le_time : time -> time -> bool possibly`

If the given times are valid, returns a `Result` that satisfies the specification of `FVTM.le_time_plain`[➦].

Errors:

- `InvalidTime` if any of the arguments is not a `FVTM.valid_time`[➦]

`val lt_time : time -> time -> bool possibly`

If the given times are valid, returns a `Result` that satisfies the specification of `FVTM.lt_time_plain`[➦].

Errors:

- `InvalidTime` if any of the arguments is not a `FVTM.valid_time`[➦]

`val unix_epoch : time`

A constant `FVTM.time`[➦] with value 1970-1-1 00:00:00, the Unix epoch.

`val utc_timestamp_plain : time -> int63`

A function which takes a `FVTM.time`[➦] and returns the number of seconds elapsed since the `FVTM.unix_epoch`[➦] in UTC to that time.

`val utc_timestamp : time -> int63 possibly`

If the given time is valid, returns a `Result` that satisfies the specification of `FVTM.utc_timestamp_plain`[➦].

Errors:

- `InvalidTime` if for the given time `t`, `FVTM.valid_time`[➦] `t` $=$ `false`

`val from_utc_timestamp_plain : int63 -> time`

A function which takes a natural number and, interpreting it as the number of seconds elapsed since the `FVTM.unix_epoch`[➦], returns the corresponding time in UTC.

`val from_utc_timestamp : int63 -> time possibly`

Given a timestamp `n`, if `n` $\leq$ 253402300826 (the maximum timestamp), returns a `Result` that satisfies the specification of `FVTM.from_utc_timestamp_plain`[➦].

Errors:

- `NumberOutOfBounds` if for the given number `n`, **not** $0 \leq$ `n` $\leq$ 253402300826

```
type weekday =
  | Monday
  | Tuesday
  | Wednesday
  | Thursday
  | Friday
  | Saturday
  | Sunday
```

A type for the days of the week.

`val weekday_of_date_plain : date -> weekday`

A function which takes a `FVTM.date`[➦] and returns its corresponding `FVTM.weekday`[➦] according to the UTC calendar.

`val weekday_of_date : date -> weekday possibly`

If the given `FVTM.date`[➦] is valid, then `FVTM.weekday_of_date`[➦] returns a `Result` that satisfies the specification of `FVTM.weekday_of_date_plain`[➦].

Errors:

- `InvalidDate` if for the given date `d`, `FVTM.valid_date`[➦] `d` = `false`

`val sort_dates_plain : date list -> date list`

A function which takes a list of dates and returns a list with the same elements in increasing order.

`val sort_dates : date list -> date list possibly`

If all the dates of the given list are valid, returns a `Result` that satisfies the specification of `FVTM.sort_dates_plain`[➦].

Errors:

- `InvalidDate` if some date of the given list is not valid

`val sort_times_plain : time list -> time list`

A function which takes a list of times and returns a list with the same elements in increasing order.

`val sort_times : time list -> time list possibly`

If all the times of the given list are valid, returns a `Result` that satisfies the specification of `FVTM.sort_times_plain`[➦].

Errors:

- `InvalidDate` if some time of the given list is not valid

`val valid_formalTime : formalTime -> bool`

A function which takes a `FVTM.formalTime`[➦] and checks if the components are on the bounds detailed in the type description:

- Component `fyear` is less than or equal to 2924712086,
- component `fmonth` is less than or equal to 35583997055,
- component `fday` is less than or equal to 1067519911673,
- component `fhour` is less than or equal to 25620477880152,
- component `fminute` is less than or equal to 1537228672809129,
- and component `fsecond` is less than or equal to 92233720368547757.

`val to_formalTime_plain : int63 -> formalTime`

A function which takes a `FVTM.int63`[➦] representing an amount of seconds and returns the same duration expressed in `FVTM.formalTime`[➦], filling the bigger components as much as possible.

`val to_formalTime : int63 -> formalTime possibly`

If the given `FVTM.int63`[➦] n satisfies n $\leq$ 92233720375632000, returns a `Result` that satisfies the specification of `FVTM.to_formalTime_plain`[➦].

Errors:

- `NumberOutOfBounds` if n \geq 92233720375632000 (365 \cdot 86400 multiplied by the maximum value for the formal year component)

```
val mkFormalTime :
  int63 ->
  int63 ->
  int63 ->
  int63 -> int63 -> int63 -> formalTime possibly
```

Produces a `Result` `FVTM.time`[➦] if the given arguments `fyears`, `fmonths`, `fdays`, `fhours`, `fminutes` and `fseconds` are valid with respect to `FVTM.valid_formalTime`[➦].

Errors:

- `NumberOutOfBounds` if the arguments do **not** satisfy `FVTM.valid_formalTime`[➦]

`val from_formalTime_plain : formalTime -> Uint63.t`

A function which takes a `FVTM.formalTime`[➦] and returns the same duration expressed in seconds.

`val from_formalTime : formalTime -> Uint63.t possibly`

If the given `FVTM.formalTime`[➦] is valid, returns a `Result` that satisfies the specification of `FVTM.from_formalTime_plain`[➦].

Errors:

- `InvalidFormalTime` if the bounds 1-6 given in `formalTime` are **not** respected

`val add_formal_plain : time -> formalTime -> time`

A function which takes a `FVTM.time`[➦] `t` and a `FVTM.formalTime`[➦] f and returns the result of adding the duration expressed by f to t.

`val add_formal : time -> formalTime -> time possibly`

If none of the error conditions below are met, returns a `Result` that satisfies the specification of `FVTM.add_formal_plain`[➦].

Errors:

- `InvalidTime` if for the given time `t`, `FVTM.valid_time`[➦] `t` = `false`
- `InvalidFormalTime` if the bounds 1-6 given in `formalTime` are **not** respected
- `Overflow` if the result would be before 1970 or after 9999

`val subtract_formal_plain : time -> formalTime -> time`

A function which takes a `FVTM.time`[➦] `t` and a `FVTM.formalTime`[➦] f and returns the result of subtracting the duration expressed by f to t.

`val subtract_formal : time -> formalTime -> time possibly`

If none of the error conditions below are met, returns a `Result` that satisfies the specification of `FVTM.subtract_formal_plain`[➦].

Errors:

- `InvalidTime` if for the given time `t`, `FVTM.valid_time`[➦] `t` = `false`
- `InvalidFormalTime` if the bounds 1-6 given in `formalTime` are **not** respected
- `Overflow` if the result would be before 1970 or after 9999

`val time_difference_plain : time -> time -> formalTime`

A function which takes two `FVTM.time`[➦] `t`, `t'` and a returns the duration elapsed between `t` and `t'` in `FVTM.formalTime`[➦].

`val time_difference : time -> time -> formalTime possibly`

If none of the error conditions below are met, returns a `Result` that satisfies the specification of `FVTM.time_difference_plain`[➦].

Errors:

- `InvalidTime` if some of the given times is invalid with respect to `FVTM.valid_time`[➦]
- `InvalidOrderOfTimes` if for the given arguments `t1`, `t2`, `FVTM.lt_time`[➦] `t1 t2` = `false`

`val sec_time_difference_plain : time -> time -> int63`

A function which takes two `FVTM.time`[➦] `t`, `t'` and a returns the duration elapsed between `t` and `t'` in seconds.

`val sec_time_difference : time -> time -> int63 possibly`

If none of the error conditions below are met, returns a `Result` that satisfies the specification of `FVTM.sec_time_difference_plain`[➦].

Errors:

- `InvalidTime` if some of the given times is invalid with respect to `FVTM.valid_time`[➦]
- `InvalidOrderOfTimes` if for the given arguments `t1`, `t2`, `FVTM.lt_time`[➦] `t1 t2` = `false`

`val shift_utc_years_plain : time -> int63 -> time`

A function which takes a time `t` and an integer n and shifts the `year` component of t the number of times determined by the signed integer n. If the result of the shifting is between 1970 and 9999 but is an invalid time, the `date_of_time` component is corrected to the closest previous valid date (if the date is invalid), and after that, the `second` component is corrected to the closest previous valid second (if the time is still invalid).

`val shift_utc_years : time -> int63 -> time possibly`

If none of the error conditions below are met, returns a `Result` that satisfies the specification of `FVTM.shift_utc_years_plain`[➦].

Errors:

- `InvalidTime` if for the given time `t`, `FVTM.valid_time`[➦] `t` = `false`
- `NumberOutOfBounds` if the result would be before 1970 or after 9999

`val shift_utc_months_plain : time -> int63 -> time`

A function which takes a time `t` and an integer n and shifts the `month` component of t the number of times determined by the signed integer n, carrying if needed to the components to the left. If the result of the shifting is between 1970 and 9999 but is an invalid time, the `date_of_time` component is corrected to the closest previous valid date (if the date is invalid), and after that, the `second` component is corrected to the closest previous valid second (if the time is still invalid).

`val shift_utc_months : time -> int63 -> time possibly`

If none of the error conditions below are met, returns a `Result` that satisfies the specification of `FVTM.shift_utc_months_plain`[➦].

Errors:

- `InvalidTime` if for the given time `t`, `FVTM.valid_time`[➦] `t` = `false`
- `NumberOutOfBounds` if the result would be before 1970 or after 9999

`val shift_utc_days_plain : time -> int63 -> time`

A function which takes a time `t` and an integer n and shifts the `day` component of t the number of times determined by the signed integer n. If the result of the shifting is between 1970 and 9999 but is an invalid time (due to leap seconds), the `second` component is corrected to the closest previous valid second.

`val shift_utc_days : time -> int63 -> time possibly`

If none of the error conditions below are met, returns a `Result` that satisfies the specification of `FVTM.shift_utc_days_plain`[➦].

Errors:

- `InvalidTime` if for the given time `t`, `FVTM.valid_time`[➦] `t` = `false`
- `NumberOutOfBounds` if the result would be before 1970 or after 9999

`val shift_utc_hours_plain : time -> int63 -> time`

A function which takes a time `t` and an integer n and shifts the `hour` component of t the number of times determined by the signed integer n, carrying if needed to the components to the left. If the result of the shifting is between 1970 and 9999 but is an invalid time (due to leap seconds), the `second` component is corrected to the closest previous valid second.

`val shift_utc_hours : time -> int63 -> time possibly`

If none of the error conditions below are met, returns a `Result` that satisfies the specification of `FVTM.shift_utc_hours_plain`[➦].

Errors:

- `InvalidTime` if for the given time `t`, `FVTM.valid_time`[➦] `t` = `false`

- `NumberOutOfBounds` if the result would be before 1970 or after 9999

`val shift_utc_minutes_plain : time -> int63 -> time`

A function which takes a time `t` and an integer n and shifts the `minute` component of t the number of times determined by the signed integer n, carrying if needed to the components to the left. If the result of the shifting is between 1970 and 9999 but is an invalid time (due to leap seconds), the `second` component is corrected to the closest previous valid second.

`val shift_utc_minutes : time -> int63 -> time possibly`

If none of the error conditions below are met, returns a `Result` that satisfies the specification of `FVTM.shift_utc_minutes_plain`[➦].

Errors:

- `InvalidTime` if for the given time `t`, `FVTM.valid_time`[➦] `t` = `false`
- `NumberOutOfBounds` if the result would be before 1970 or after 9999

`val shift_utc_seconds_plain : time -> int63 -> time`

A function which takes a time `t` and an integer n and shifts the `second` component of t the number of times determined by the signed integer n, carrying to the components on the left if needed. In other words, the function adds n seconds to the time t.

`val shift_utc_seconds : time -> int63 -> time possibly`

If none of the error conditions below are met, returns a `Result` that satisfies the specification of `FVTM.shift_utc_seconds_plain`[➦].

Errors:

- `InvalidTime` if for the given time `t`, `FVTM.valid_time`[➦] `t` = `false`
- `NumberOutOfBounds` if the result would be before 1970 or after 9999

`val add_formal_seconds_plain : time -> int63 -> time`

A function which takes a `FVTM.time`[➦] `t` and a `FVTM.int63`[➦] n and returns the result of adding n formal seconds to t.

`val add_formal_seconds : time -> int63 -> time possibly`

If none of the error conditions below are met, returns a `Result` that satisfies the specification of `FVTM.add_formal_seconds_plain`[➦].

Errors:

- `InvalidTime` if for the given time `t`, `FVTM.valid_time`[➦] `t` = `false`
- `NumberOutOfBounds` if the result would be before 1970 or after 9999

`val add_formal_minutes : time -> int63 -> time possibly`

A function which takes a `FVTM.time`[➦] `t` and a `FVTM.int63`[➦] n and returns the result of adding n formal minutes to t.

Errors:

- `InvalidTime` if for the given time `t`, `FVTM.valid_time`[➦] `t` = `false`

- `NumberOutOfBounds` if the integer argument `n` $\leq$ -76861433640456466 (minimum `FVTM.int63`[➡] divided by the formal minute duration) or if the integer argument `n` $>$ 76861433640456465 (maximum `FVTM.int63`[➡] divided by the formal minute duration), or if the result would be before 1970 or after 9999

`val add_formal_hours : time -> int63 -> time possibly`

A function which takes a `FVTM.time`[➡] `t` and a `FVTM.int63`[➡] n and returns the result of adding n formal hours to t.

Errors:

- `InvalidTime` if for the given time `t`, `FVTM.valid_time`[➡] `t` = `false`
- `NumberOutOfBounds` if the integer argument `n` $\leq$ -1281023894007608 (minimum `FVTM.int63`[➡] divided by the formal hour duration) or if the integer argument `n` $>$ 1281023894007607 (maximum `FVTM.int63`[➡] divided by the formal hour duration), or if the result would be before 1970 or after 9999

`val add_formal_days : time -> int63 -> time possibly`

A function which takes a `FVTM.time`[➡] `t` and a `FVTM.int63`[➡] n and returns the result of adding n formal days to t.

Errors:

- `InvalidTime` if for the given time `t`, `FVTM.valid_time`[➡] `t` = `false`
- `NumberOutOfBounds` if the integer argument `n` $\leq$ -53375995583651 (minimum `FVTM.int63`[➡] divided by the formal day duration) or if the integer argument `n` $>$ 53375995583650 (maximum `FVTM.int63`[➡] divided by the formal day duration), or if the result would be before 1970 or after 9999

`val add_formal_months : time -> int63 -> time possibly`

A function which takes a `FVTM.time`[➡] `t` and a `FVTM.int63`[➡] n and returns the result of adding n formal months to t.

Errors:

- `InvalidTime` if for the given time `t`, `FVTM.valid_time`[➡] `t` = `false`
- `NumberOutOfBounds` if the integer argument `n` $\leq$ -1779199852789 (minimum `FVTM.int63`[➡] divided by the formal month duration) or if the integer argument `n` $>$ 1779199852788 (maximum `FVTM.int63`[➡] divided by the formal month duration) or if the result would be before 1970 or after 9999

`val add_formal_years : time -> int63 -> time possibly`

A function which takes a `FVTM.time`[➡] `t` and a `FVTM.int63`[➡] n and returns the result of adding n formal years to t.

Errors:

- `InvalidTime` if for the given time `t`, `FVTM.valid_time`[➡] `t` = `false`
- `NumberOutOfBounds` if the integer argument `n` $\leq$ -146235604339 (minimum `FVTM.int63`[➡] divided by the formal year duration) or if the integer argument `n` $>$ 146235604338 (maximum `FVTM.int63`[➡] divided by the formal year duration), or if the result would be before 1970 or after 9999