A1-D10-2

Implementation: The CTTN-System, Version 2007

Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Project number:	IST-2004-506779
Project instrument:	EU FP6 Network of Excellence (NoE)
Project thematic priority:	Priority 2: Information Society Technologies (IST)
Document type:	D (deliverable)
Nature of document:	R (report)
Dissemination level:	PU (public)
Document number:	IST506779/Munich/A1-D10-2/D/PU/a1
Responsible editors:	H.J. Ohlbach
Reviewers:	
Contributing participants:	Munich
Contributing workpackages:	A1
Contractual date of deliverable:	29 February 2008
Actual submission date:	29 March 2008

Abstract
The CTTN–system is a computer program which provides advanced processing of temporal notions. The basic data structures of the CTTN–system are time points, crisp and fuzzy time intervals, labelled partitionings of the time line, granularities, and calendar systems. The labelled partitionings and granularities are used to model periodic temporal notions, quite regular ones like years, months etc., partially regular ones like timetables, but also very irregular ones like, for example, dates of a conference series. These data structures can be used in the temporal specification language GeTS (GeoTemporal Specifications). GeTS is a functional specification and programming language with a number of built-in constructs for specifying customised temporal notions.

CTTN is implemented as a Web server and as a C++ library. This document gives a detailed overview over the current state of the system and its components.

Keyword List
temporal notions

Project co-funded by the European Commission and the Swiss Federal Office for Education and Science within the Sixth Framework Programme.

Implementation: The CTTN-System, Version 2007

Hans Jürgen Ohlbach

Department of Computer Science, University of Munich
Email: `ohlbach@ifi.lmu.de`

29 March 2008

Abstract
The CTTN–system is a computer program which provides advanced processing of temporal notions. The basic data structures of the CTTN–system are time points, crisp and fuzzy time intervals, labelled partitionings of the time line, granularities, and calendar systems. The labelled partitionings and granularities are used to model periodic temporal notions, quite regular ones like years, months etc., partially regular ones like timetables, but also very irregular ones like, for example, dates of a conference series. These data structures can be used in the temporal specification language GeTS (GeoTemporal Specifications). GeTS is a functional specification and programming language with a number of built-in constructs for specifying customised temporal notions.

CTTN is implemented as a Web server and as a C++ library. This document gives a detailed overview over the current state of the system and its components.

Keyword List
temporal notions

Contents

Chapter 1

A General Overview Over the CTTN–System

1.1 Introduction

In the CTTN–project we aim at a very detailed modelling of the temporal notions. These are, in particular, time points, crisp and fuzzy temporal intervals together with built-in as well as user definable relations between and operations on these intervals. Furthermore, there is support for various kinds of regular and irregular periodic temporal notions, again built-in ones as well as user definable ones. The possibilities range from very simple ones like seconds or minutes up to complex ones like Easter time or solar eclipses. A special specification and programming language GeTS (GeoTemporal Specifications [32]) allows applications and users to defined their own versions of temporal notions and to do all kinds of computations with them.

CTTN is *not* the implementation of a theoretical temporal logic, but it models the flow of time as it is perceived on our planet. It realizes the main concepts and operations underlying many temporal notions in natural language.

The key components of the CTTN–system consist of the modules depicted in Figure 1.1. The Service module at the bottom contains a large variety of application independent functions. The FuTI module (Fuzzy Time Intervals) [31, 34] contains the data structures and operations on time time points and crisp and fuzzy time intervals. The largest module is the PartLib module (Partitioning module). It contains the machinery for specifying and working with periodic temporal notions. Since calendar systems consist of such periodic temporal notions, a module for representing different calendar systems is also part of PartLib.
The GeTS–module implements a functional programming language with certain additional constructs for this application area. A flex/bison type parser and an abstract machine for GeTS has been implemented as part of the CTTN–system. GeTS is the first specification and programming language with such a rich variety of built-in data structures and functions for GeoTemporal notions. In a first case study it has been used to define various versions of fuzzy interval–interval relations [34].

The basic interface to the CTTN –system is socket based and implements the CTTN protocol. Prototypes of RMI, CORBA and SOAP interfaces have also been implemented, but not yet fully tested.

1.2 Time Points and Time Intervals in the FuTI–Module

The flow of time underlying most calendar systems corresponds to a time axis which is isomorphic to the real numbers $\mathbb{R}$. Since the most precise clocks developed so far, atomic clocks, measure the time in discrete units, it is sufficient to restrict the representation of concrete time points to *integers*. Therefore FuTI represents time points with integers, either with 64–bit integers, or with multiple precision integers (this is a compiler option). Within FuTI there is no assumption about the meaning of these integers, whether they are days, seconds, femtoseconds or not even time points[1].

Although FuTI represents time points only with integers, there is still the underlying assumption that the time axis is isomorphic to the real numbers. That means, for example, the interval between the time points 0 and 1 is not empty, but it is set of real numbers between 0 and 1.

[1]A special component of FuTI, which was developed for another application allows for the representation of circular intervals like angles between 0 and 360 degrees. In this case the integers represent fractions of angular degrees.

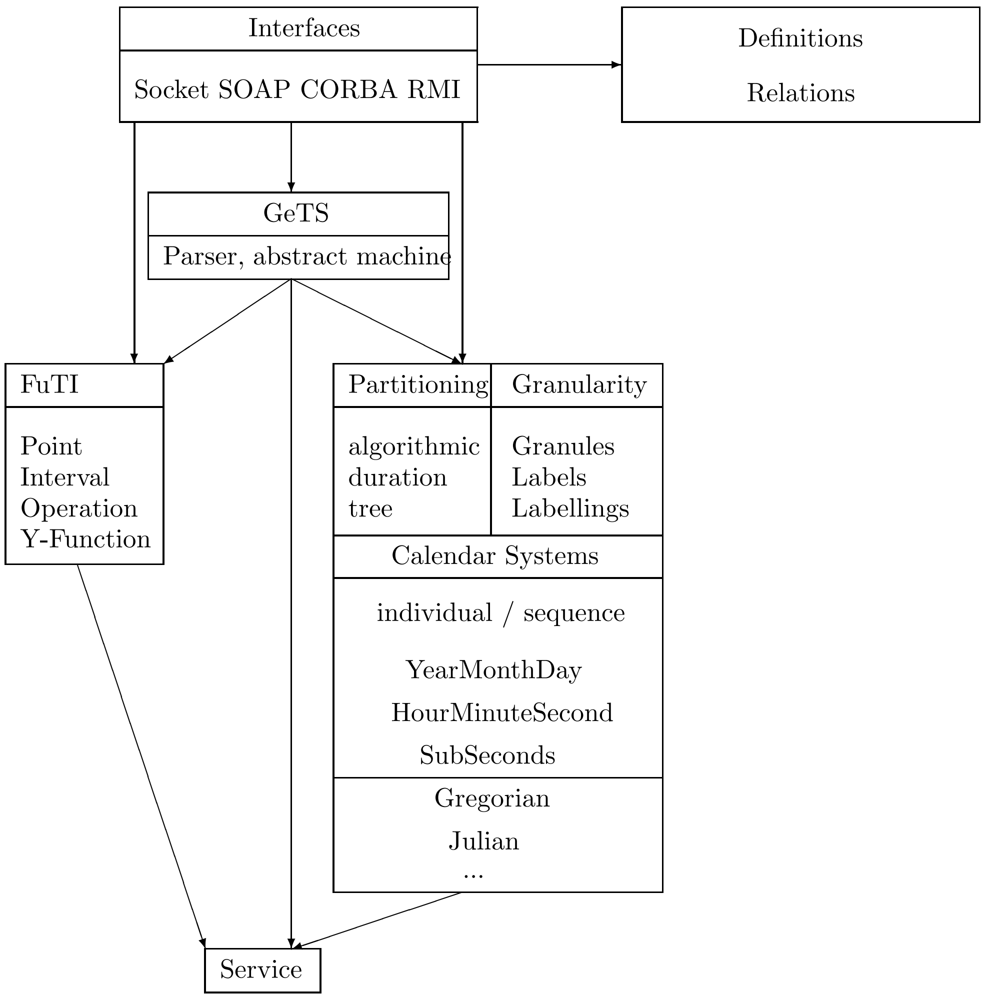
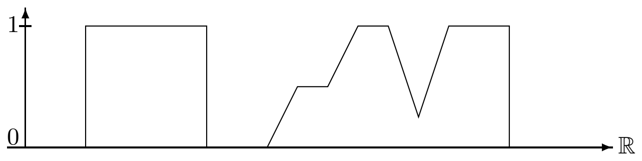


Figure 1.1: The CTTN-System

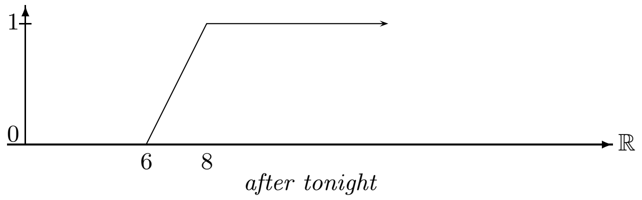
The next important data type is that of time intervals. Time intervals can be crisp or fuzzy. With fuzzy intervals one can encode notions like ‘around noon’ or ‘late night’ etc. Since fuzzy intervals are more general and more flexible than crisp intervals, FuTI uses fuzzy intervals as basic interval data type.

Fuzzy intervals are usually defined through their membership functions [41, 14]. A membership function maps a base set to real numbers between 0 and 1. The base set for fuzzy time intervals is a linear time axis.



Crisp and Fuzzy Intervals

The fuzzy intervals can also be infinite. For example, the term 'after tonight' may be represented as a fuzzy distribution which rises from fuzzy value 0 at 6 pm until fuzzy value 1 at 8 pm and then remains 1 ad infinitum.



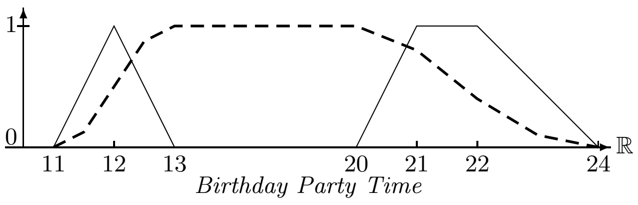
after tonight

Fuzzy time intervals are realized in the FuTI–module as polygons with integer coordinates. The x-coordinates represent time points and the y-coordinates represent fuzzy values as integers between 0 and a maximum value (the default value is 1000). A normalised fuzzy value between 0 and 1 can then be obtained by dividing the integer y-coordinate by the maximum value. A y-coordinate of 500, for example, represents the normalised fuzzy value 0.5.

If the integers represent hours, one can, for example, represent the interval 'around noon' as the polygon ((11,0) (12,1000) (13,0)). The membership function of the corresponding fuzzy interval starts at 11 o'clock with fuzzy value 0 and then rises linearly to fuzzy value 1 at noon. From there on it falls linearly to fuzzy value 0 at 1 pm.

FuTI provides a large collection of operations on these intervals. There are methods for accessing information about the intervals, the location of various parts of an interval, its size (which is the integral over the membership function), its components etc. There are methods for transforming the intervals, for example, hull computations. There are integration functions, fuzzification functions etc. There are also very general unary and binary transformation functions which can be parameterised with functions operating on the fuzzy values. All the set operations on fuzzy intervals, for example, are realized as transformations with functions on the fuzzy values. The transformations of the fuzzy membership functions need not be linear, i.e. they may transform straight lines into curved lines. The FuTI–module contains for these cases an approximation algorithm which approximates curved lines by polygons.

Example 1.2.1 (Birthday Party Time) This example illustrates some of the operations which are possible with the FuTI–module. Consider the statement "the birthday party for took place *from around noon until early evening* of 20/7/2003". The corresponding fuzzy interval could be generated by integrating the fuzzy interval for 'around noon' in positive direction, integrating the fuzzy interval for 'early evening' in negative direction and then intersecting the two integrals. The resulting fuzzy set is:



A GeTS–specification of this example is given in Example 1.4.5. ■

1.3 Periodic Temporal Notions in the PartLib–Module

The PartLib–module offers powerful machinery for specifying and working with periodic temporal notions. The basic concept is the concept of the *partitionings of the time axis*. Since most periodic temporal notions, for example, days, yield infinite partitionings of the time axis, PartLib offers different versions of finite representations of these infinite structures. The operations on the infinite structures are turned into operations on the corresponding finite representations.

Partitions can be *labelled*, e.g., with ‘Monday’, ‘Tuesday’ etc. Partitionings with labels can be comprised in different ways to different structures. For example, from the day–partitioning and the corresponding labelling one can derive the structure which corresponds to ‘all Mondays’ or to ‘all non-Mondays’. If the labels are organised in a hierarchy, for example, Monday,..,Friday are all ‘Workdays’ and Saturday and Sunday are ‘Weekenddays’ one can derive the notion of ‘all Workdays’. Since there are a number of further ways to derive new substructures of the time axis from labelled partitionings, all these ways are comprised into the concept of *granularities* (see Sec. 1.3.5). A granularity is essentially a subset of a partitioning of the time axis. Many operations in the CTTN–system work with the more general concept of granularities instead with partitionings.

1.3.1 Partitionings of the Time Axis

Most basic time units of calendar systems, years, months etc., are essentially partitionings of the time axis. Other periodical temporal notions, for example, semesters, school holidays, sunsets and sunrises etc., can also be modelled as partitionings.

A partitioning of the real numbers $\mathbb{R}$ may be, for example, $(..., [-100, 0[, [0, 100[, [100, 101[, [101, 500[, ...)$. The intervals in the partitionings need not be of the same length (because time units like years are not of the same length either). The intervals can, however, be enumerated by integers (their *coordinates*). For example, we could have the following enumeration

$$\begin{array}{cccccc} ... & [-100\ 0[& [0\ 100[& [100\ 101[& [101\ 500[& ... \\ ... & -1 & 0 & 1 & 2 & ... \end{array}$$

The enumeration of partitions, i.e. their coordinates, are a very useful means for concrete computations. It turned out, however, that in some cases instead of integer coordinates, certain other structures which are isomorphic to integers are more useful. An example for a structure which is isomorphic to the integers are the paths in an infinite tree. Therefore PartLib has introduced the concept of *Integer Structures* as a generalisation of the integer coordinates.

Definition 1.3.1 (Partitioning) *A* partitioning P *of the time axis in PartLib is a sequence*

$$\ldots [t_{-1}, t_0[, [t_0, t_1[, [t_1, t_2[, \ldots$$

of non-empty half open intervals in $\mathbb{R}$ with integer boundaries such that $t_i < t_{i+1}$ for all i.

The partitioning may be finite at one or both sides, i.e. $]-\infty, t_0[, ..., [t_n, +\infty[$ *is allowed.*

An Integer Structure *is a set of objects which is isomorphic to the integers.*

A coordinate mapping *c is a bijective mapping between a partitioning and an Integer Structure (or a part of it if the partitioning is finite) such that if partition p is before partition q then $c(p) < c(q)$.* ■

The choice of half open intervals of the kind $[t_i, t_{i+1}[$ as partitions was arbitrary. It means that, for example, Midnight always belongs to the next day.

1.3.2 Labelled Partitionings

The partitions in CTTN can be *labelled*. The labels are just names for the partitions like in the following example.

Example 1.3.2 (The Labelling of Days) We count the time in seconds beginning with January 1st 1970. This was a Thursday. Therefore we choose as labelling for the day partitioning

$$L \stackrel{\text{def}}{=} Th, Fr, Sa, Su, Mo, Tu, We.$$

The following correspondences are obtained:

time :	...	$[-86400, 0[$	$[0, 86400[$	$[86400, 172800[$	...
coordinate :	...	-1	0	1	...
label :	...	We	Th	Fr	...

This means, for example, $Label(-1) = We$, i.e. December 31 1969 was a Wednesday. ■

Labels are different to coordinates because different partitions can have the same label (e.g., all Mondays). Labellings can be used for three purposes. The first purpose is to get access to the partitions via their names (labels). One can use these names in various GeTS–functions. The second purpose is to associate partitions with further attributes. The labels can, for example, serve as keys into databases. The third purpose is to use the labels for grouping partitions together into *granules*. An example is the set of all Mondays. This is no longer a partitioning of the time axis because there are gaps between the Mondays.

Definition 1.3.3 (Labels) *A set of* labels *in PartLib is just an arbitrary finite or infinite set*[2]

A label hierarchy *is a binary relation $\sqsubseteq$ which orders the labels in a tree.*

A labelling *of a PartLib partitioning is a possibly partial mapping from the partitions into the set of labels.* ■

Since a labelling can be partial, not all partitions need to have labels. As an example, where this makes sense, consider the partitioning of hours and the labelling which associates the label 'working hour' with all hours between 8 am and noon and all hours between 1 pm and 5 pm. The other hours don't have labels. This labelling specifies implicitly the concept of 'working day', the concept of 'lunch time', and the concept of 'after work'. These implicit definitions can be made explicit in PartLib by turning them into *granularities* (see below).

[2]Labels are in fact instances of subclasses of a class *Label*.

1.3.3 Specification of Partitionings

Partitionings have a finite representation in PartLib. There are the following representations for partitionings.

Algorithmic Partitionings
This type of partitionings is mainly used for modelling the basic time units of calendar systems, years, months etc. The specification consists of an offset against time point 0, an average length of the partitions, and a correction function which corrects the average length to the actual length. Algorithmic Partitionings are useful for modelling very basic time units, such as months or years, but also for very complex ones, such as Easter time, sunrises and sunsets, tides etc.

Duration Partitionings
They are specified by an anchor time and a sequence of *'durations'*.

For example, I could define 'my weekend' as a *duration partitioning* with anchor time 2004/7/23, 4 pm (Friday July, 23rd, 2004, 4 pm) and durations: ('8 hour + 2 day', '4 day + 16 hour'). The first interval would be labelled 'weekend'.

A simpler example is the notion of a semester at a university. In the Munich case, the dates could be: anchor time: October 2000. The durations are: 6 months (with label 'winter semester') and 6 months (with label 'summer semester'). This defines a partitioning with partition 0 starting at the anchor time, and then extending into the past and the future. The first partition in this example is the winter semester 2000/2001.

The units for the duration are in fact granularities, and not just partitionings. Thus, one can, for example, define durations in terms of *granules.* An example is '3.5 working_days + 1.5 weekends'.

Date Partitionings
In this version we provide the boundaries of the partitions by concrete dates. Therefore the partitioning can only cover a finite part of the time line.

An example could be the dates of the Time conferences: 1994/5/4 Time94 1994/5/4 gap 1995/4/26 Time95 1995/4/26 . . . 2004/7/1 Time04 2004/7/3.

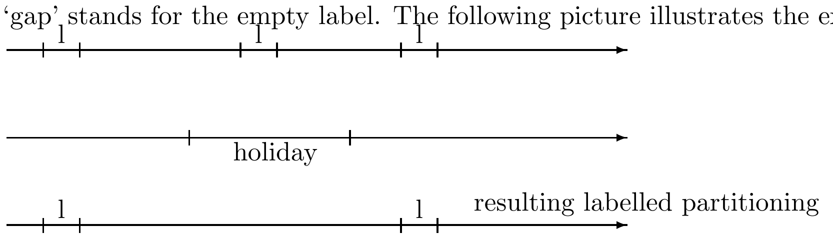
Since the intervals between two adjacent dates determine durations, date partitionings are in fact special cases of duration partitionings, and this is one of the two possibilities how they are treated in PartLib. The other possibility is to generate an explicit representation of the finitely many partition boundaries.

Intersection Partitionings
They combine two previously defined partitionings by intersecting their partitions. If the two original partitionings are labelled then a new labelling can be computed by means of *mapping rules* for labels.

As an example, suppose there is a partitioning $p1$ representing the lecture course l, say every Wednesday from 10 am until 12 am. There is a second partitioning $p2$ which represents public holidays. $p2$ is labelled with the holiday names (Easter, Christmas etc.) The holiday name labels are all sub-labels in a label hierarchy with top element 'holiday'. The partitioning which represents the lecture time without the public holidays can be generated by intersecting $p1$ and $p2$ with the following mapping rules

$$\begin{array}{lll} l * holiday & \mapsto & gap; \\ l * gap & \mapsto & l; \\ gap * holiday & \mapsto & gap \end{array}$$

with the extra provision that adjacent partitions without labels are comprised into a single partition. 'gap' stands for the empty label. The following picture illustrates the example.



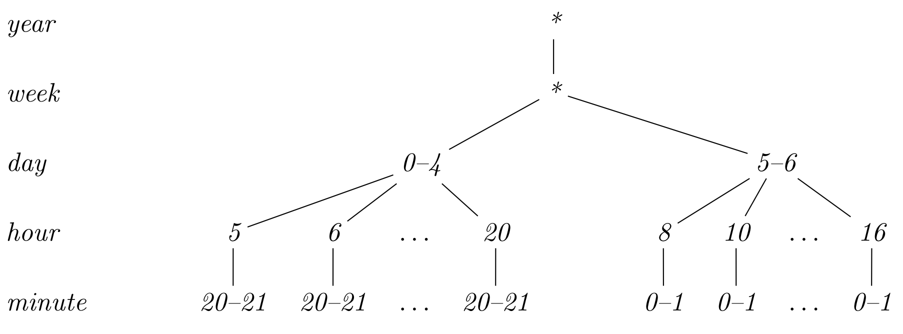
Shifted Partitionings
Suppose a bus timetable for a particular bus station has been defined as a partitioning. The bus time table for the next bus station may be such that all time are just shifted by, say, 5 minutes. It would be very inconvenient to force a user to specify this new timetable in the same complicated way as the first one. Moreover, if the original bus timetable is changed, the timetables for all other stations have to be changed as well. A much easier and safer way would be to define the other timetables by taking the first one and just shifting it by a certain duration.

To this end a partitioning type *Shifted Partitioning* has been introduced. It is specified by a given partitioning and a duration. All partitions of the new partitioning, together with their labels, are generated by shifting the original partitions by the given duration.

Tree Partitionings (TPS)
This type of specification for partitionings can be used when concrete dates are involved. Typical examples are bus timetables. A tree partitioning is given by a *Date Format* and a *Range Tree.* The date format determines a kind of calendar to be used for interpreting the nodes in the range tree.

Example 1.3.4 (for a Tree Partitioning Specification) *A typical date format is the standard date format year/week/day/hour/minute/second.*

The following TPS may define a bus schedule.



It specifies the following bus schedule: every year, every week, every work day (0–4), there is a bus at 5:20 – 5.21 (2 minutes stay at the bus stop), 6:20 – 6:21 until 20:20 – 20:21, and at the weekends (days 5,6) there is a bus every hour from 8 until 16 hours.

The nodes in the TPS determine an offset from the start of the granule given by the corresponding position in the date format. There are four different node types:

NumberRange nodes. They specify concrete number sets, for example, 4-6,10-12 specifies the set $\{4, 5, 6, 10, 11, 12\}$

NumberIterator nodes. They specify iterators like, for example, in a 'for loop'. The iterator is given by a start value, a step value and a number of iterations. For example, start = 1, step = 2, iterations = 5 specifies the set $\{1, 3, 5, 7, 9\}$

LabelRange nodes. They specify concrete label sets, for example, March-May, August specifies the set $\{2, 3, 4, 7\}$ (January is month 0).

LabelIterator nodes. They specify labels by giving a label together with a number iterator. For example, Label = 'L', start = 2, steps = 10, iterations = 5 The loop starts with the second occurrence of L and then continues 5 times in steps of 10 partitions with this label, 5 iterations.

In all four cases it is also possible to interpret the numbers as distances from the end of a partition. For example, if the day partition is right below the month partition in the corresponding date format, and the backwards flag is set to true, then the number 0 at the day level is interpreted as the very last day in the given month.

The specification of a partitioning can be quite complex and require a lot of data. Therefore for each partitioning type, except for algorithmic partitionings, there is a corresponding XML document type for specifying a partitioning. After the CTTN interface has read and parsed such an XML specification one can use them in the same way as the built-in partitionings for calendar systems.

1.3.4 Leap Seconds

To compensate for the slowing down of the earth's rotation, since 1971 every few years a leap second has been introduced. The last minute in the year where a leap second has been inserted has 61 seconds instead of 60 seconds. This has an effect on all partitionings above the level of seconds. It would be very complicated and error prone to integrate the effect of leap seconds in all these partitionings. As an alternative, this phenomenon is taken care of by separating the reference time into a *global reference time* and a *local reference time*. The global reference time counts the seconds as they are. It knows nothing about leap seconds. The local reference time shrinks the leap seconds to 0 length. That means the last minute in the years where a leap second has been inserted has still 60 seconds in the local reference time. The extra second occurs only in the transition to the global reference time. This way the leap second calculations have been concentrated in a single place, the transition between local and global reference time. All other partitioning dependent calculations can ignore leap seconds.

1.3.5 Granularities

The labels which can be attached to the partitionings generate a variety of new substructures of the time axis which are no longer partitionings because there can be gaps between the corresponding time intervals. Since periodic temporal notions with gaps are much more frequent than partitionings, the concept of *granularities* has been introduced.

Granularities are like partitionings, but there are two essential differences

- there are gaps allowed between two neighbouring granules
- there are gaps allowed even within a granule. An example is 'working day' from 8 am until 5 pm with a lunch break from 12 am until 1 pm.

CTTN distinguishes the following types of granularities:

Partitioning Granularities: each partition is a granule. Labels are ignored.

Label Partition Granularities: are determined by a label (possibly within a label hierarchy). For example, the Label Partition Granularity with label 'weekendday' of a day partitioning (with sub-labels Saturday and Sunday below weekendday) would join the days of the weekends into a granule. A Saturday is a granule, followed by the following Sunday, followed by the following Saturday etc.

Label Block Granularities: is similar to a Label Partition Granularity. The difference is that neighbouring partitions with the given label form one granule. A Label Block Granularity with labels 'weekendday' (see above) would join Saturdays and Sundays into one single granule.

Labelling Granularities: declares a whole label sequence as a granule. For example, the labelling 'Monday', 'Tuesday', ... 'Sunday' of the day partitioning comprises a whole week into a single granule.

Gap Block Granularities: A Gap Block Granularity comprises all adjacent partitions without labels into one granule.

As soon as a labelling has been attached to a partitioning, all these types of granularities are available as concrete data types, and a common API is available via the superclass 'Granularity'. Typical examples for the API are methods which move from a given granule to the next granule, methods which move from a given time point n granules forward or backward (n may be fractional), methods which measure time intervals in terms of granule length etc.

1.3.6 Calendar Systems

A *calendar system* in the CTTN–system is a set of partitionings or granularities, for example the partitionings for seconds, minutes, hours, weeks, months and years, together with some extra data and methods. Dershowitz and Reingold's 'calendrical calculations' are used here [13] for computing the details down to the level of days. In addition PartLib models all the nasty features of real calendar systems, in particular leap seconds and daylight saving time schemes (in a submodule *DLST*). Calendar systems can be arranged in sequences, for example, the sequence consisting of the Julian calendar system until 4th of October 1582 followed by the Gregorian system. Another example of a sequence of calendar systems in PartLib could be a sequence of calendars and time zones a traveller encounters when he travels around the world.

The Calendar submodule in PartLib has predefined general classes for years/ months/days, for hours/minutes/seconds and for sub-seconds. Using these classes it requires very little code to add new calendar systems.

1.4 The GeTS–Language

The PartLib–module has, via the XML-interface, mechanisms for integrating user defined periodic temporal notions. Not all temporal notions and computations, however, have to do with periodicies. The GeoTemporal Specification Language GeTS has therefore been added as a general purpose language for working with temporal notions. The design of the GeTS–language was influenced by the following considerations:

1. Although the GeTS–language has many features of a functional programming language, it is not intended as a general purpose programming language. It is a specification language for temporal notions, however, with a concrete operational semantics.

2. The parser, compiler, and in particular the underlying GeTS abstract machine are not standalone systems. They must be embedded into a host system which provides the data structures and algorithms for time intervals, partitionings etc., and which serves as the interface to the application. GeTS provides a corresponding application programming interface (API).

3. The language should be simple, intuitive, and easy to use. It should not be cluttered with too many features which are mainly necessary for general purpose programming languages.

4. The last aspect, but even more the point before, namely that GeTS is to be integrated into a host system, were the main arguments against an easy solution where GeTS is only a particular module in a functional language like SML or Haskell. The host system was developed in C++ (it could also be Java, but multiple precision integers are more efficient in C++). Linking a C++ host system to an SML or Haskell interpreter for GeTS would be more complicated than developing GeTS in C++ directly. The drawback is that features like sophisticated type inferencing or general purpose data structures like lists or vectors are not available in the current version of GeTS.

5. Developing GeTS from scratch instead of using an existing functional language has also an advantage. One can design the syntax of the language in a way which better reflects the semantics of the language constructs. This makes it easier to understand and use. As an example, the syntax for a time interval constructor is just $[expression_1, expression_2]$.

The GeTS–language is a strongly typed functional language with a few imperative constructs. Here we can give only a flavour of the language. The technical details are in Chapter 4 Comment[32].

Example 1.4.1 (tomorrow) The definition

```
tomorrow = partition(now(),day,1,1)
```

specifies 'tomorrow' as follows: `now()` yields the time point of the current point in time. `day` is the name of the day partitioning. Let i be the coordinate of the day-partition containing `now()`. `partition(now(),day,1,1)` computes the interval $[t_1, t_2[$ where t_1 is the start of the partition with coordinate $i + 1$ and t_2 is the end of the partition with coordinate $i + 1$. Thus, $[t_1, t_2[$ is in fact the interval which corresponds to 'tomorrow'.

In a similar way, we can define

```
this_week(Time t)  = partition(t,week,0,0).
```

The time point `t`, for which the week is to be computed, is now a parameter of the function. ■

Example 1.4.2 (Christmas) The definition

```
christmas(Time t) =
  dLet year = date(t,Gregorian_month) in
                 [time(year|12|25,Gregorian_month),
                  time(year|12|27,Gregorian_month)]
```

specifies Christmas for the year containing the time point `t`. ■

`date(t,Gregorian_month)` computes a date representation for the time point `t` in the date format `Gregorian_month` (`year/month/day/hour/minute/second`). Only the year is needed. `dLet year = ...` therefore binds only the year to the integer variable `year`. If, for example, in addition the month is needed one can write `dLet year|month = date(....`

`time(year|12|25,Gregorian_month)` computes t_1 = begin of the 25th of December of this year. `time(year|12|27,Gregorian_month)` computes t_2 = begin of the 27th of December of this year. The expression `[...,...]` denotes the half open interval $[t_1, t_2[$.[3] The result is therefore the half open interval from the beginning of the 25th of December of this year until the end of the 26th of December of this year.

Example 1.4.3 (Point–Interval Before Relation) The function

```
PIRBefore(Time t, Interval I) =
     if (isEmpty(I) or isInfinite(I,left)) then false
     else (t < point(I,left,support))
```

specifies the standard crisp point–interval 'before' relation in a way which works also for fuzzy intervals. ■

If the interval `I` is empty or infinite at the left side then `PIRBefore(t,I)` is `false`, otherwise `t` must be smaller than the left boundary of the support of `I`.

Now we define a parameterised fuzzy version of the interval–interval before relation.

Example 1.4.4 (Fuzzy Interval–Interval Before Relation) A fuzzy version of an interval–interval before relation could be

```
IIRFuzzyBefore(Interval I, Interval J, Interval->Interval B) =
case
  isEmpty(I) or isEmpty(J) or
       isInfinite(I,right) or isInfinite(J,left)     : 0,
  (point(I,right,support) <= point(J,left,support))  : 1,
   isInfinite(I,left) : integrateAsymmetric(intersection(I,J),B(J))
else integrateAsymmetric(I,B(J))
```

■

The input are the two intervals `I` and `J` and a function `B` which maps intervals to intervals. `B` is used to compute for the interval `J` an interval `B(J)`, which represents the degree of 'beforeness' for the points before `J`.

The function first checks some trivial cases where `I` cannot be before `J` (first clause in the `case` statement), or where `I` definitely is before `J` (second clause in the `case` statement). If `I` is infinite at the left side then $\int (I \cap J)(x) \cdot B(J)(x)dx/|I \cap J|$ is computed to get a degree of 'beforeness', at least for the part where I and J intersect. If `I` is finite then $\int I(x) \cdot B(J)(x)dx/|I|$ is computed. This averages the degree of a point–interval 'beforeness', which is given by the product $I(x) \cdot B(J)(x)$, over the interval `I`.

[3]Crisp intervals in CTTN are always half open intervals $[\ldots,\ldots[$. Sequences of such intervals, for example, sequences of days, can therefore be used to partition a time period. The syntactic representation of these intervals in GeTS is `[...,...]` and not `[...,...[` because this simplifies the grammar and the parser considerably.

The next example is a parameterised version of an 'Until' operator. It can be used to formalise expressions like 'from around noon until early evening'. The parameters are operators which manipulate the front and back end of the intervals, together with a complement operator.

Example 1.4.5 (Until) *an 'Until' operator can be defined in GeTS:*

```
Until(Interval I, Interval J, Side s1, Side s2,
      (Interval*Interval)->Interval Ints,
      Interval->Interval Ep, Interval->Interval En,
      Interval->Interval C) =
        if (s1 == left) then
           (if (s2 == left) then Ints(Ep(I),C(Ep(J)))
                            else Ints(Ep(I),En(J)))
        else
           (if (s2 == left) then Ints(C(En(I)),C(Ep(J)))
                            else Ints(C(En(I)),En(J)));
```

The birthday party example (Example 1.2.1) could be specified using this function:

```
Birthdayparty(I,J)
     = Until(I, J, left, right,
       lambda(Interval K, Interval L) intersection(K,L),
       lambda(Interval K) integrate(K,positive),
       lambda(Interval K) integrate(K,negative),
       lambda(Interval K) complement(K)).
```

■

1.5 The Web–Interface

CTTN is a collection of C++ classes and methods which can be used in any other C++ program. There is, however, also a command interface which is realized as a web server. It communicates with a client through a socket. There is a group of commands for uploading application specific definitions of temporal notions in the GeTS–language and in the specification language for labelled partitionings. There are also commands for working with instances of these temporal notions, particular time intervals, particular partitionings, particular calendar systems etc.

More details about the CTTN–system are available at the CTTN homepage:
http://www.pms.ifi.lmu.de/CTTN.

1.6 Notation

A typical mathematical notation is something like "an object of type X is an n-tuple $x = (a_1, \ldots, a_n)$ where a_1 is a ..., a_2 is a ..., and ...". If the a_i themselves are some m-tuples, it becomes sometimes quite cumbersome to express which component is actually meant. This is made much simple with a notation which comes from object oriented programming. The components of the n-tuples become *names* and for a particular n-tuple x they are accessed by $x.\langle name \rangle$.

As an example, suppose we have a concept "PersonName" with components (first name, middle name, surname). For a particular object $x =$ (John, F., Kennedy), we could write

$x.FirstName$ to access "John", x.MiddleName for "F." and x.Surname for "Kennedy". If we have another object type, for example "presidency" with components PersonName and country, we could iterate the notation and write $y.PersonName.FirstName$ to access the component "John" of the object $y =$ ((John, F., Kennedy),USA)

The notation can be extended to denote functions (methods) operating on these objects. For example, if there is a function *StringRepresentation* for a name, we could write

$$y.PersonName.\mathit{StringRepresentation}()$$

to get a string representation for the given name. The functions can of course have additional parameters. If *StringRepresentation*, for example, has an additional Boolean parameter "upper case", we could write

$$y.PersonName.\mathit{StringRepresentation}(\mathit{true})$$

to get the string representation with upper case letters.

Overloaded Functions There are many cases where there are a number of functions which deliver the same information, but which have different argument types. In this case one can use the same (*overloaded*) function names. The type of the argument determines the concrete version of the function. This is a common notation in mathematics, where one uses, for example, the same + function in $x+y$ and the type of x and y (integer, real or complex numbers, vectors etc.) determine the meaning of the x.

Some more or less standard mathematical notations are further listed in Def. 1.6.1 below.

Definition 1.6.1 (Notation) *We use the following notations:*

$\mathbb{I}$	*is the set of integers*
$\mathbb{N}$	*is the set of natural numbers with 0*
$\mathbb{R}$	*is the set of real numbers*
$\mathbb{R}^+$	$\overset{\text{def}}{=} \mathbb{R} \cup \{-\infty, +\infty\}$.

$\sup\{s\}$ *is the* supremum *of the set* $s \subseteq \mathbb{R}$ *and*
$\inf\{s\}$ *is the* infimum *of the set* $s \subseteq \mathbb{R}$.
For an interval $I = [a, b] \subseteq \mathbb{R}$ *let* $|I| \overset{\text{def}}{=} b-a$ *be the length of* I. *If* I *consists of several subintervals let* $|I|$ *be the sum of the length of the subintervals. The same definitions apply if* I *consists of open or half-open intervals.*

If $x \in \mathbb{R}$ *then* $\lfloor x \rfloor$ *is the integer part of* x, *e.g.* $\lfloor 3.5 \rfloor = 3$ *and* $\lfloor -3.5 \rfloor = -3$.

If L *is an ordered list of length* $|L|$ *and* $i \in \{0, \ldots, |L| - 1\}$ *we write* $L[i]$ *to denote the* i*'th element of the list.*

$min_i\{f(i) \mid \varphi(i) \mid m(i)\}$ *yields the value* $f(i)$ *for the parameter* i *for which the condition* $\varphi(i)$ *holds and for where* $m(i)$ *is minimised.*

$max_i\{f(i) \mid \varphi(i) \mid m(i)\}$ *yields the value* $f(i)$ *for the parameter* i *for which the condition* $\varphi(i)$ *holds and for where* $m(i)$ *is maximised.* ■

An example for the *max* construct is:

$$max_i\{age(i) \mid i \text{ is a person} \mid hight(i)\}$$

yields the age of the tallest person.

Definition 1.6.2 (Time) *We define the data types* `Time` *and* $\texttt{Time}_\infty$ *with infinities:*

$$\begin{aligned} \texttt{Time} &\overset{\text{def}}{=} \mathbb{I} \\ \texttt{Time}_\infty &\overset{\text{def}}{=} \mathbb{I} \cup \{-\infty, +\infty\} \end{aligned}$$

If $s = -\infty$ *and* $t \in \texttt{Time}$ *then* $[s, t[$ *stands for* $]-\infty, t[$. ■

Structure of the Document. Chapter 2 contains a description of the FuTI–module (fuzzy intervals). Chapter 3 contains a description of the PartLib module (for periodic temporal notions). Chapter 4 contains a description of the GeTS–language. An application of the GeTS– language where fuzzy temporal relations for fuzzy intervals are defined is presented in Chapter 5. The appendix contains the API of the FuTI module (Appendix A) and the syntax description of the GeTS–language (Appendix B).

Chapter 2

Fuzzy Time Intervals – The FuTI–Module

Abstract: The FuTI–module is a collection of classes and methods for representing and manipulating fuzzy time intervals. The chapter consists of three parts. The mathematical theory is described in the first part. The concrete representation of fuzzy time intervals as polygons, together with the algorithms for operating on these polygons is presented in the second part. Finally, the third part contains a short description of the application interface. Version 1.1 contains in addition a class for representing circular fuzzy intervals which can, for example, be used to represent fuzzy angles. FuTI is an open source C++–module. .

2.1 Motivation and Introduction

Many temporal notions used in everyday life have a deliberate imprecise meaning. For example, if I say in the morning "tonight I'll go to the disco", and somebody asks me "will you go to the disco at 8 pm?" I may neither want to say "yes" nor may I want to say "no". One may argue whether in this case any precise mathematical model of "tonight" is useful at all. There are other cases, however, where a fuzzy logic model of imprecise notions is definitely helpful. Consider, for example, a database with, say, a cinema timetable. If you query the timetable "give me all performances ending *before* midnight", do you really want to exclude a performance ending just one minute after midnight? I think, not. One could solve this problem by giving the 'before' relation a fuzzy meaning, such that performances ending before midnight get a fuzzy value 1, and performances ending after midnight get a fuzzy value which decreases the later the performance ends. The fuzzy value could then be used to order the answers to the query such that the performances ending after midnight come late in the list.

In this paper the FuTI–module (Fuzzy Temporal Intervals) of data structures and algorithms for representing and manipulating fuzzy temporal notions is described. In the first part the components of the FuTI–module are described in a purely mathematical way, without any commitment to concrete data structures and algorithms. A representation of the fuzzy intervals as polygons with integer coordinates is explained in the second part. All algorithms in FuTI work on these polygons. Finally the concrete interface to the module is listed and explained.

The fuzzy intervals in FuTI are fuzzy subsets of the real values. Therefore they can represent all kinds of things. The main motivation for most of the operations in FuTI, however, comes from their interpretation as *temporal* intervals and relations; and this is the reason for the 'T' in FuTI.

2.2 The Mathematics of Fuzzy Time Intervals

The mathematics of general fuzzy sets [41] has been investigated in great depth. The particular fuzzy sets in FuTI are subsets of the real numbers. On the one hand, this makes things easier. On the other hand, however, it offers a very rich algebraic structure with many different operations and relations. Therefore it is useful to start with an overview of the basic ideas and definitions about fuzzy sets. Some, but not all of them can be found in textbooks about fuzzy sets (see e.g. [14]).

Since FuTI is designed as a library to be used in many different applications, we need to provide a broad spectrum of quite different concepts and operations. I tried to organise them in a meaningful way and to motivate them with temporal notions and operations.

2.2.1 Fuzzy Time Intervals

Fuzzy Intervals are usually defined through their membership functions. A membership function maps a base set to a real number between 0 and 1. This "fuzzy value" denotes a kind of degree of membership to a fuzzy set S. For example, the base set may consist of all people on earth, and S may be the set of 'large persons'. If for the person John the fuzzy value for 'large persons' is 1 then John is definitely a large person. If the fuzzy value is 0 then John is definitely not a large person. If, instead, the fuzzy value is, for example, 0.8, then John is quite tall, but not as tall as really large persons.

The base set for fuzzy time intervals is the time axis. In FuTI it is represented by the set $\mathbb{R}$ of real numbers. Real numbers allow us to model the continuous time flow which we perceive in our life. A fuzzy time interval in FuTI is now a fuzzy subset of the real numbers.

Definition 2.2.1 (Fuzzy Time Intervals) *A* fuzzy membership function *in FuTI is a total function $f : \mathbb{R} \mapsto [0,1]$ which need not be continuous, but it must be integratable.*

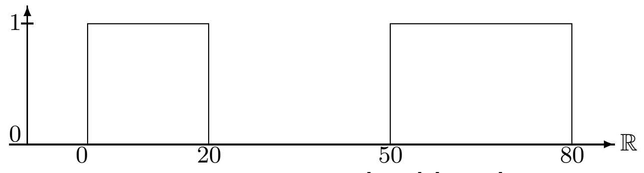
The fuzzy interval I_f *that corresponds to a fuzzy membership function f is*

$$I_f \stackrel{\text{def}}{=} \{(x,y) \subseteq \mathbb{R} \times [0,1] \mid y \leq f(x)\}.$$

Given a fuzzy interval I we usually write $I(x)$ to indicate the value of the corresponding membership function at point x.

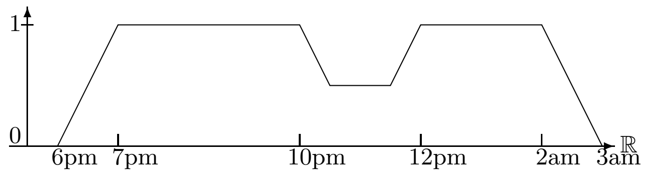
Let $F_{\mathbb{R}}$ be the set of fuzzy time intervals. ■

This definition comprises single or multiple crisp intervals like this:



Crisp Fuzzy Intervals [0,20],[50,80]

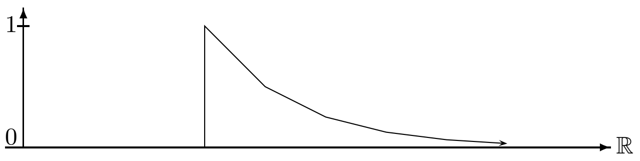
It also comprises finite fuzzy intervals like this one:



Party Time

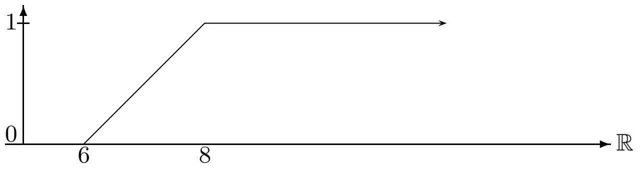
This set may represent a particular party time, where the first guests arrive at 6 pm. At 7 pm all guests are there. Half of them disappear between 10 and 12 pm (because they go to the pub next door to watch an important soccer game). Between 12 pm and 2 am all of them are back. At 2 am the first ones go home, and finally at 3 am all are gone. The fuzzy value indicates in this case the number of people at the party.

Fuzzy intervals may also be infinite.



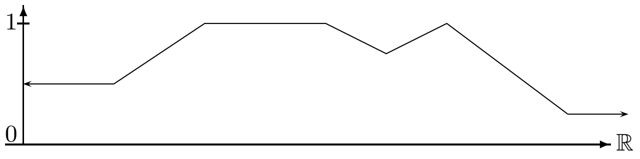
Danger Time: Caused by Radioactive Decay

More realistic examples of infinite fuzzy time intervals are intervals where the fuzzy value remains constant after a while. For example, the term 'after tonight' may be represented by a fuzzy interval which rises from fuzzy value 0 at 6 pm until fuzzy value 1 at 8 pm and then remains 1 ad infinitum.



After Tonight

The more general case are infinite fuzzy intervals which may be infinite at one or two sides, but where the membership function becomes constant, but not necessarily 1, after a while.



Infinite Fuzzy Interval with Mostly Constant Membership Function

Remark 2.2.2 *The representation of a fuzzy interval as a subset of* $\mathbb{R} \times [0,1]$ *means that one can apply the standard set operators* $\cup$ *(union),* $\cap$ *(intersection) and* $\setminus$ *(set difference) to fuzzy intervals. The standard set theoretic definition of intersection and set difference, however, need not yield fuzzy intervals any more. Therefore there are other, more appropriate definitions of the set operations on fuzzy intervals (see Section 2.2.4).*

Definition 2.2.3 (Height of a Fuzzy Interval (sup, inf)) *For a fuzzy interval* $I \in F_{\mathbb{R}}$ *let*

$$\sup(I) \overset{\text{def}}{=} \sup\{I(x) \mid x \in \mathbb{R}\}$$

be the height *(largest fuzzy value, supremum) of* I *and let*

$$\inf(I) \overset{\text{def}}{=} \inf\{I(x) \mid x \in \mathbb{R}\}$$

be the smallest fuzzy value*, (infimum) of* I ■

$\sup(I)$ is usually, but not necessarily, 1 for nonempty fuzzy time intervals. If $\sup(I) = 0$ then, however, I must be empty.

Fuzzy time intervals may be quite complex structures with many different characteristic features. The simplest ones are *core* and *support.* The core is the subset of $\mathbb{R}$ where the fuzzy value is 1, and the support is the subset of $\mathbb{R}$ where the fuzzy value is non-zero. In addition, we define the *kernel* as the subset of $\mathbb{R}$ where the fuzzy value is *not* constant ad infinitum. Finally, *maxRegion* is the interval between the first and last point where the fuzzy value is maximal.

Definition 2.2.4 (Core, Support, Kernel and MaxRegion)
The core $C(I)$ *of a fuzzy set* I *is the subset of* $\mathbb{R}$ *where the membership function is 1:*

$$C(I) \overset{\text{def}}{=} \{x \in \mathbb{R} \mid I(x) = 1\}.$$

The core of I *can be empty even if* I *itself is not empty.*

The support $S(I)$ *of* I *is the subset of* $\mathbb{R}$ *where the membership function is nonzero:*

$$S(I) \overset{\text{def}}{=} \{x \in \mathbb{R} \mid I(x) \neq 0\}.$$

If $S(I) = \emptyset$ *then* $I = \emptyset$.

The kernel $K(I)$ *of* I *is the smallest interval* $[a, b[\subseteq \mathbb{R}^+$ *such that there are* $I_1 \in [0,1]$ *and* $I_2 \in [0,1]$ *with* $I(x) = I_1$ *for all* $x < a$ *and* $I(x) = I_2$ *for all* $x > b$.

$K(I)$ *can be empty, finite or infinite. If* $K(I) = \emptyset$ *then* I *is either empty or infinite with at most two different fuzzy values.*

The maxRegion $M(I)$ *of* I *is the interval between the first and last maximal points, i.e.*

$$M(I) \overset{\text{def}}{=} \begin{cases} [\inf\{x \mid I(x) = \sup(I)\}, \sup\{x \mid I(x) = \sup(I)\}[& \textit{if } sup(I) \neq 0 \\ [] & \textit{otherwise} \end{cases}$$

For $O \in \{C, S, K, M\}$ *let* $O^{\sqcap}(I)$ *be the (crisp) fuzzy interval such that* $C(O^{\sqcap}(I)) = S(O^{\sqcap}(I)) = O^{\sqcap}(I)$.

For $O \in \{C, S, K, M\}$ *let*

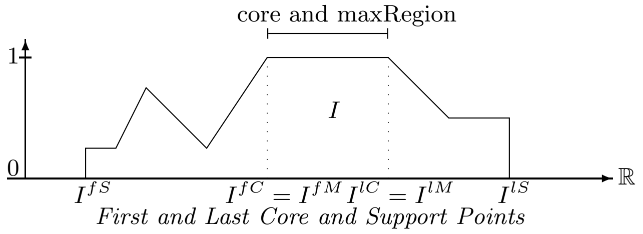
$$I^{fO} \overset{\text{def}}{=} \begin{cases} \inf\{x \mid O(I)(x) \neq 0\} & \textit{if } O(I) \neq \emptyset \\ +\infty & \textit{otherwise} \end{cases}$$

be the first O-point *of* I *and let*

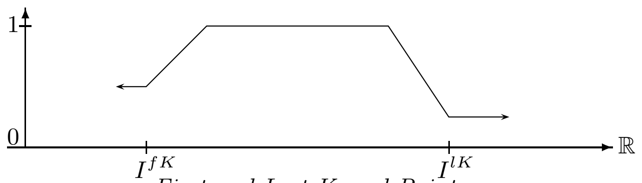
$$I^{lO} \overset{\text{def}}{=} \begin{cases} \sup\{x \mid O(I)(x) \neq 0\} & \textit{if } O(I) \neq \emptyset \\ +\infty & \textit{otherwise} \end{cases}$$

be the last O-point *of* I. ■

I^{fC} and I^{lC} are the first/last core points.
I^{fS} and I^{lS} are the first/last support points.
I^{fK} and I^{lK} are the first/last kernel points.
I^{fM} and I^{lM} are the first/last maxRegion points.

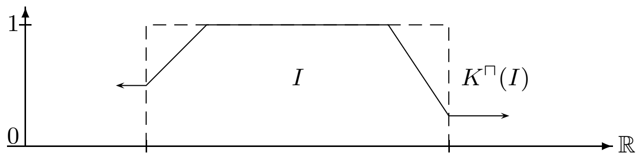


First and Last Core and Support Points



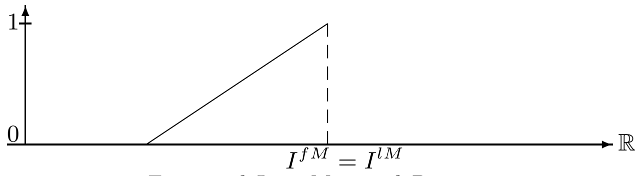
First and Last Kernel Points

The next picture shows the kernel of the same interval I as crisp interval $K^{\sqcap}(I)$.



$K^{\sqcap}(I)$

The next picture shows an example where $I^{fM} = I^{lM}$ and where $\sup(I)$ is really the supremum, and not the maximum because $I(I^{fM}) = 0$.



First and Last Maximal Points

If $\sup(I) = 1$ then $I^{fM} = I^{fC}$ and $I^{lM} = I^{lC}$. If, however, $\sup(I) < 1$ then I^{fM} and I^{lM} have nothing to do with the core of I.

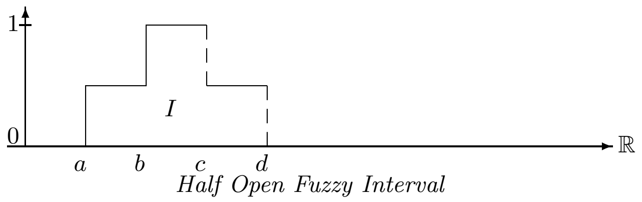
Fuzzy time intervals with finite kernel are of particular interest because although they may be infinite, they can easily be implemented with finite data structures. Therefore we give them an extra name.

Definition 2.2.5 (Fuzzy Time Intervals with Finite Kernel) *Let $F^f_{\mathbb{R}}$ be the set of fuzzy time intervals (Def. 2.2.1) with finite kernel (Def. 2.2.4).* ■

Fuzzy time intervals which are in fact crisp intervals can now be characterised very easily as intervals where core and support are the same.

Definition 2.2.6 (Crisp Interval) *A* crisp interval *is a fuzzy interval I (Def. 2.2.1) such that $C(I) = S(I)$ (Def. 2.2.4).* ■

Remark 2.2.7 (Openness and Closedness) *Ordinary intervals can be open or closed. A similar distinction can also be made for fuzzy intervals. As an example, consider the following fuzzy interval I:*



Half Open Fuzzy Interval

If we have $I(a) = 0.5$, $I(b) = 1$, but $I(c) = 0.5$ and $I(d) = 0$ then I is closed at a and b and open at c and d.

We sometimes indicate the open sides of fuzzy intervals with dashed lines. ■

Half-open intervals are of particular interest for time intervals. Consider, for example, the two intervals 'this week's Monday' and 'this week's Tuesday'. If both intervals are represented as closed intervals then midnight belongs to Monday and Tuesday. This is not what we usually want. Therefore it is more realistic to represent the intervals as half-open intervals such that midnight belongs to either Monday or Tuesday, but not to both days. As a convention, we assume that (finite) time intervals are half open at the positive side: their structure is $[a, b[$. Midnight would then belong to Tuesday. This has some consequences for the algorithms (cf. Remark 2.3.24).

2.2.2 Scalar Properties of Fuzzy Time Intervals

Fuzzy time intervals can be measured in various ways. Besides the size, which is the integral over the membership function, one can locate the position of the core, support and kernel. One can also measure the maximal fuzzy value. This should, but need not be 1. Furthermore, one can split the interval into parts of equal size (the first half and the second half etc.), and locate their boundaries. Let us start with the size of the interval.

Definition 2.2.8 (Size) *For $a, b \subseteq \mathbb{R}^+$ and a fuzzy time interval I let*

$$|I|_a^b \overset{\text{def}}{=} \int_a^b I(x)\ dx. \qquad |I| \overset{\text{def}}{=} |I|_{-\infty}^{+\infty} \text{ is the size of } I.$$

■

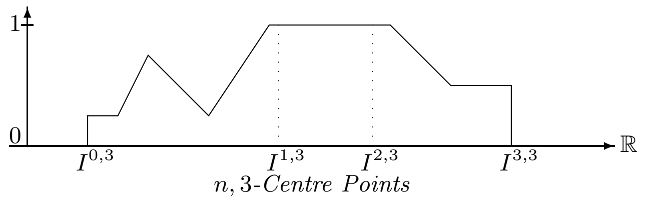
If I is a crisp interval then $|I|$ yields the length of I in the usual sense.

Centre Points

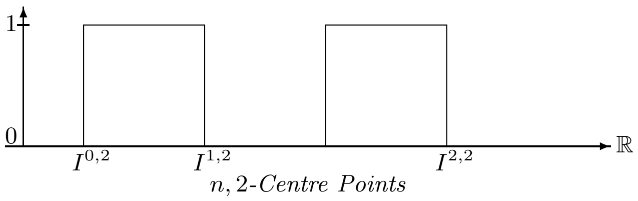
The n, m*-centre points* defined below are used to express temporal notions like 'the first half of the year', or 'the second quarter of the year' or more exotic expressions like 'the 25th 49th of the weekend' etc. The notion of n, m-centre points makes only sense for finite intervals.

Definition 2.2.9 (n, m-Centre Points) *Let $I \in F_{\mathbb{R}}$ with $|I| < \infty$. For two integers $m > 1$ and $0 \leq n \leq m$ we define the n, m-*centre points
$I^{n,m} \overset{\text{def}}{=} x_n$ *where x_n is a minimal $\mathbb{R}$-value in a sequence $I^{fS} = x_0, \ldots, x_m = I^{lS}$ with $|I|_{x_0}^{x_1} = |I|_{x_1}^{x_2} = \ldots = \ldots |I|_{x_{m-1}}^{x_m} = |I|/m$.* ■

Examples: $I^{1,2}$ splits I in two halfs of the same size. $I^{1,3}$ indicates a split of I into three parts of the same size. $I^{1,3}$ is the boundary of the first third, $I^{2,3}$ is the boundary of the second third.



$n, 3$-Centre Points



n, 2-Centre Points

Middle Points:
The middle point between the centre points $I^{n,m}$ and $I^{n+1,m}$ is just $I^{2n+1,2m}$. For example, the middle point in the first half of I is $I^{1,4}$ and the middle point in the second half is $I^{3,4}$.

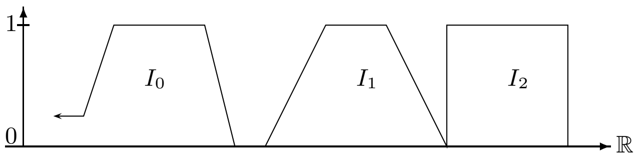
Components
Fuzzy time intervals can consist of several different components. A component is a sub-interval of a fuzzy interval such that the left and right end is either the infinity, or the membership function drops down to 0. Let $Cmp(I)$ be the list of components of I. $nComponents(I)$ is the number of components of I. $component(I, k)$ is the k^{th} component of I.

Definition 2.2.10 (Components) *Let $I \in F_{\mathbb{R}}$. The* components $I_0, \ldots, I_n$ *of I are fuzzy time intervals such that: (i) $I_k(x) = I(x)$ for all $x \in S(I_k)$ and $0 \leq k \leq n$, and (ii) for all $k \in \{1, \ldots, n-1\}$:* $(\lim_{x \to I_k^{fs}} I(x) = 0$ *or* $\lim_{I_k^{fs} \leftarrow x} I(x) = 0)$ *and* $(\lim_{x \to I_k^{ls}} I(x) = 0$ *or* $\lim_{I_k^{ls} \leftarrow x} I(x) = 0)$.

Let $nComponents(I)$ be the number of components of I.
Let $component(I, k)$ be the k^{th} component of I. ■

The definition is quite complicated because we want to count as separate components parts of fuzzy time intervals where the membership function drops down to 0 at just one single point.

Example:



Components

2.2.3 Functions Operating on Fuzzy Time Intervals

Time intervals usually don't appear from nowhere, but they are constructed from other time intervals. We distinguish two ways of constructing new fuzzy time intervals, first by means of *y-functions* and then by means of *interval operators*. Y-functions map fuzzy values to fuzzy values. They can therefore be used to construct a new interval from a given one by applying the y-function point by point to the membership function values.

Interval operators are more general construction functions. They take one or more fuzzy time intervals and construct a new one out of them.

Definition 2.2.11 (Y-Functions)
Y-FCT$^n \overset{\text{def}}{=} \{f : [0,1]^n \mapsto [0,1]\}$ *is the set of n-place* y-functions.
They map fuzzy values to fuzzy values.
Y-FCT$\overset{\text{def}}{=} \bigcup_{n\geq 0}$ *Y-FCT*^{n}. ■

Definition 2.2.12 (Interval Operators)
I-OPs$^n \overset{\text{def}}{=} \{g : F_{\mathbb{R}}{}^n \mapsto F_{\mathbb{R}}\}$ *is the set of n-place* interval operators.
They map fuzzy intervals to fuzzy intervals.
I-OPs$\overset{\text{def}}{=} \bigcup_{n\geq 0}$ *I-OPs*^{n}. ■

Every y-function can be used to construct a new fuzzy time interval from given ones by applying the y-function to the fuzzy values.

Definition 2.2.13 (Associated Interval Operators) *If* $f \in$ *Y-FCT*^{n} *is a y-function then* $g_f \in$ *I-OPs*^{n} *defined by* $g_f(I_1, \ldots, I_n)(x) \overset{\text{def}}{=} f(I_1(x)), \ldots, I_n(x))$ *is the associated interval operator.* ■

Linear Y-Functions
A small, but important class of y-functions are *linear* y-functions. They are important firstly because very natural operators, like standard complement, intersection and union of fuzzy time intervals can be described with linear y-functions. Secondly they are important because they allow us to transform intervals represented by polygons in a very efficient way: only the vertices and intersection points of the polygons need to be transformed.

The main characterisation of linear y-functions is therefore that they map non-intersecting straight line segments to straight line segments, and not to curves.

Definition 2.2.14 (Linear Y-Function) *A y-function* $f \in$ *Y-FCT*^{n} *is linear if the mapping* $f'((x, y_1), \ldots, (x, y_n))\overset{\text{def}}{=}(x, f(y_1, \ldots, y_n))$ *maps non-intersecting line segments* $(x_1, z_{11}) - (x_2, z_{12})$, $\ldots$, $(x_1, z_{n1}) - (x_2, z_{n2})$ *to a line segment* $(x_1, f(z_{11}, \ldots, z_{n1})) - (x_2, f(z_{12}, \ldots, z_{n2}))$. ■

One-place linear y-functions can be characterised in the following way:

Proposition 2.2.15 (Characterisation of One-Place Linear y-Functions) *A one-place y-function* f *is linear if and only if* $f(y) = f(0) + (f(1) - f(0)) \cdot y$ *holds.*

Proof: Suppose f is linear. We take the straight line segment between $(0,0)$ and $(1,1)$. The mapping $f'(x,y) \overset{\text{def}}{=} (x, f(y))$ maps this line segment to a line segment between $(0, f(0))$ and $(1, f(1))$. Therefore

$$\begin{aligned} f(y) &= f(0) + \tfrac{f(1)-f(0)}{1-0} \cdot (y-0) \quad \text{(line equation)} \\ &= f(0) + (f(1) - f(0)) \cdot y \end{aligned}$$

The other direction of the proof is trivial. ■

An example for a one-place linear y-function is the standard negation $n(y) = 1 - y$.

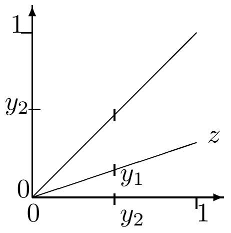
The characterisation of two-place linear y-functions is a bit trickier.

Proposition 2.2.16 (Characterisation of Two-Place Linear y-Functions) *A two-place y-function* f *is linear if and only if the following condition holds:*

$$f(y_1, y_2) = \begin{cases} f(0,0) + (f(y_1/y_2, 1) - f(0,0)) \cdot y_2 & \textit{if } y_1 \leq y_2 \\ f(0, (y_1 - y_2)/(1-y_2)) + (f(1,1) - f(0, (y_1-y_2)/(1-y_2))) \cdot y_2 & \textit{otherwise} \end{cases}$$

Proof: Suppose f is linear.

We consider the case $y_1 \leq y_2$ first. To this end we take the straight line segment between $(0,0)$ and $(1,1)$. The line equation for this line is just $y = x$. Now take an arbitrary $y_2 \in [0,1]$ and an arbitrary $y_1 \leq y_2$. The line equation for the line segment starting at $(0,0)$ and crossing (y_2, y_1) is $y = (y_1 - 0)/(y_2 - 0) \cdot x$. For $x = 1$ we get $z = y_1/y_2$.

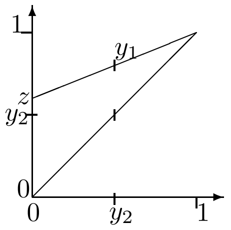


Since f is linear we have

$$\begin{aligned} f(y_1, y_2) &= f(0,0) + \tfrac{f(z,1) - f(0,0)}{1-0} \cdot y_2 \\ &= f(0,0) + (f(\tfrac{y_1}{y_2}, 1) - f(0,0)) \cdot y_2 \end{aligned}$$

Now consider the case $y_1 \geq y_2$.

The line starting at $(1,1)$ and crossing (y_2, y_1) crosses the y-axis at $z = (y_1 - y_2)/(1 - y_2)$.



Since f is linear we have

$$\begin{aligned} f(y_1, y_2) &= f(0,z) + \tfrac{f(1,1) - f(0,z)}{1-0} \cdot y_2 \\ &= f(0, \tfrac{y_1 - y_2}{1 - y_2}) + (f(1,1) - f(0, \tfrac{y_1 - y_2}{1 - y_2})) \cdot y_2 \end{aligned}$$

The other direction, showing that the two conditions imply linearity, is again straightforward. ■

Simple examples for linear two-place y-functions are the minimum and maximum functions. The minimum function is used to realize standard intersection of two fuzzy time intervals, and the maximum function is used to realize standard union of two fuzzy time intervals.

2.2.4 Set Operators for Fuzzy Intervals

For ordinary intervals there are the standard Boolean set operators: complement, intersection, union etc. These are uniquely defined. There is no choice. Unfortunately, or fortunately, because it gives you more flexibility, there are no such uniquely defined set operators for fuzzy intervals. Set operators are essentially transformations of the membership functions, and there are lots of different ones. One has tried to classify them such that essential properties of the Boolean set operators are preserved.

Complement of Fuzzy Time Intervals
The complement operator for fuzzy time intervals is to be understood in the following sense: if for a particular point x the probability to belong to a set S is y then the probability to belong to the complement of S is $n(y)$ where n is a so called *negation function.*

Definition 2.2.17 (Negation Function) *A function $n \in$ Y-FCT^1 satisfying the conditions*

- $n(0) = 1$ *and* $n(1) = 0$*;*
- n *is non-increasing, i.e.* $\forall x, y \in [0,1] : x \leq y \Rightarrow n(x) \geq n(y)$

is called a negation function*.*

Let NF be the set of all negation functions*.* ■

Example 2.2.18 (Standard Negation and λ-Complement) *The function*

$$n(y) \overset{\text{def}}{=} 1 - y$$

is the standard fuzzy negation.

For any $\lambda > -1$ the so called λ-complement is the function

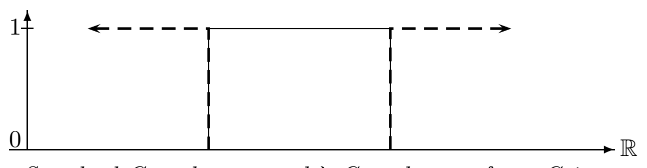
$$n_\lambda(y) \overset{\text{def}}{=} \frac{1-y}{1+\lambda y}.$$

Both functions n and n_λ are negation functions in the sense of Def. 2.2.17.
$N(I)(x) \overset{\text{def}}{=} n(I(x))$ is the standard complement *operator.*
$N_\lambda(I)(x) \overset{\text{def}}{=} n_\lambda(I(x))$ is the λ-complement *operator.*
If I is a crisp interval then $N(I) = N_\lambda(I)$ ■

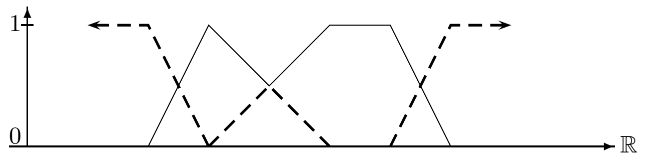
Proposition 2.2.19 (Idempotency of the negation functions) *For every $y \in [0,1]$ we have for the standard negation $n(n(y)) = y$ and for the λ-complement: $n_\lambda(n_\lambda(y)) = y$.*
The proof is straightforward. ■

This property need not hold for other negation functions.

We give some examples for standard and λ-complement. The dashed lines indicate the complement.

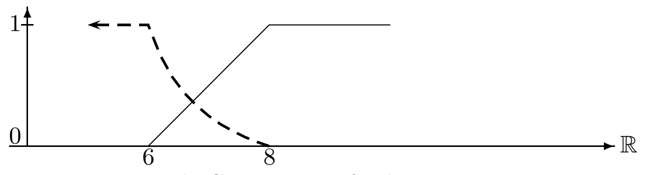


Standard Complement and λ-Complement for a Crisp Interval



Standard Complement for a Fuzzy Interval

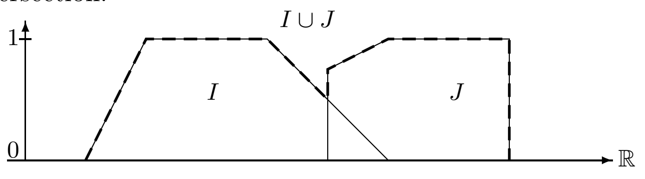
If we define 'tonight' as a fuzzy interval, rising from 0 at 6pm to 1 at 8pm, we could use the standard complement for 'before tonight'. The term 'long before tonight' must of course be represented differently to 'before tonight'. A λ-complement version with $\lambda = 2$ looks as follows:



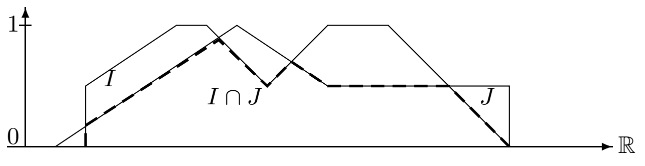
λ-Complement for $\lambda = 2$

If λ is increased then the descend from 6 pm till 8 pm becomes steeper. A suitable λ could then in fact mean 'long before tonight'.

Intersection and Union of Fuzzy Time Intervals
The two figures below show standard union and intersection of fuzzy intervals. Union is obtained by taking the maximum of the two member functions. The minimum of the two member functions yields intersection.



Standard Union of Fuzzy Sets



Standard Intersection of Fuzzy Sets

Standard union and intersection, however, is only one particular form of union and intersection. Instead of minimum and maximum one could think of other functions for computing union and intersection. These other functions, so called triangular norms and co-norms, must fulfil certain axioms in order to satisfy our intuition about union and intersection.

Definition 2.2.20 (Triangular Norms and Co-norms) *A function* $T : [0,1]^2 \mapsto [0,1]$ *is called a* triangular norm*, or* t-norm *for short, iff it satisfies the laws T1-T4 below. A function* $S : [0,1]^2 \mapsto [0,1]$ *is called a* triangular co-norm*, or* t-co-norm *for short, iff it satisfies the laws S1-S4 below.*

Identity law*:*	**T1:**	$\forall x$	$T(x,1) = x,$
	S1:	$\forall x$	$S(x,0) = x$
Commutativity*:*	**T2:**	$\forall x,y$	$T(x,y) = T(y,x),$
	S2:	$\forall x,y$	$S(x,y) = T(y,x)$
Associativity*:*	**T3:**	$\forall x,y,z$	$T(x,T(y,z)) = T(T(x,y),z),$
	S3:	$\forall x,y,z$	$S(x,S(y,z)) = S(S(x,y),z)$

Monotonicity*:* $\forall x,y,u,v \in [0,1]\ x \leq u, y \leq v :$

T4:		$T(x,y) \leq T(u,v),$
S4:		$S(x,y) \leq S(u,v)$

Triangular norms and co-norms are y-functions in Y-FCT^2 *(Def. 2.2.11).*
Let $TNorm$ *be the set of triangular norms and*
let $TCoNorm$ *be the set of triangular co-norms.* ■

The triangular norms and co-norms are now turned into interval operators $\cap_T$ and $\cup_S$:

Definition 2.2.21 (Intersection and Union) *Let* $I, J \in F_{\mathbb{R}}$ *be two fuzzy intervals. If* T *is a triangular norm and* S *a triangular co-norm (Def. 2.2.20) then*

$$(I \cap_T J)(x) \stackrel{\text{def}}{=} T(I(x), J(x)) \qquad \text{and} \qquad (I \cup_S J)(x) \stackrel{\text{def}}{=} S(I(x), J(x))$$

are the intersection and union operators on the fuzzy intervals. ■

Example 2.2.22 (Standard Fuzzy Intersection and Union) *The function* min *is a triangular norm and the function* max *is a triangular co-norm. Therefore*

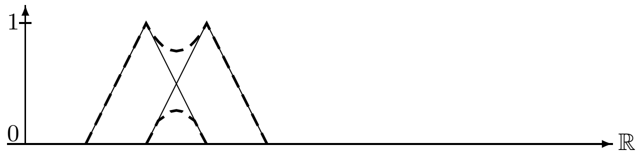
$$(I \cap_{\min} J)(x) \overset{\text{def}}{=} \min(I(x), J(x)) \qquad \textit{and} \qquad (I \cup_{\max} J)(x) \overset{\text{def}}{=} \max(I(x), J(x))$$

are the standard fuzzy intersection and union operators. ■

A particular class of triangular norms and co-norms, together with a negation function, is the *Hamacher family.*

Example 2.2.23 (Hamacher Family) *The Hamacher family consists of the following parameterised families of triangular norms and co-norms, and negation functions (λ-complement):*

$$\begin{aligned} T_\gamma(x,y) &\overset{\text{def}}{=} \frac{xy}{\gamma + (1-\gamma)(x+y-xy)} && \gamma \geq 0 \\ S_\beta(x,y) &\overset{\text{def}}{=} \frac{x+y+(\beta-1)xy}{1+\beta xy} && \beta \geq -1 \\ n_\lambda(x) &\overset{\text{def}}{=} \frac{1-x}{1+\lambda x} && \lambda > -1 \end{aligned}$$

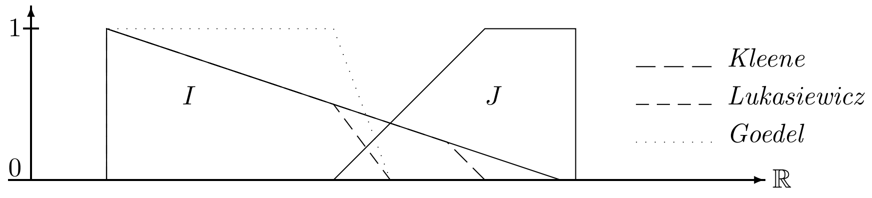


Hamacher Intersection and Union with $\beta = \gamma = 0.5$ ■

Set Difference of Fuzzy Time Intervals Set difference $I \setminus J$ can also be defined by means of y-functions. The following versions are derived from corresponding implication functions:

Definition 2.2.24 (Set Difference)
Kleene: $(I \setminus J)(x) \overset{\text{def}}{=} min(I(x), 1 - J(x))$
Lukasiewicz: $(I \setminus J)(x) \overset{\text{def}}{=} max(0, I(x) - J(x))$ ■
Goedel: $(I \setminus J)(x) \overset{\text{def}}{=} 0$ *if* $I(x) \leq J(x)$ *and* $1 - J(x)$ *otherwise*

Example 2.2.25 (Set Difference) *The following picture shows the difference between the three versions of set difference.*



Set Difference $I \setminus J$

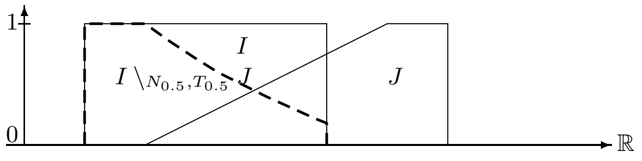
■

The Kleene version corresponds to the crisp definition of set difference: $I \setminus J = I \cap J^c$ where J^c is the complement of J. This can be generalised by replacing $\cap$ with $\cap_T$ and J^c with a complement operator.

Definition 2.2.26 (Generalised Set Difference) *Let N be a complement function and T a t-norm. We define the set difference operator $\setminus_{N,T}$ between two fuzzy intervals I and J as*

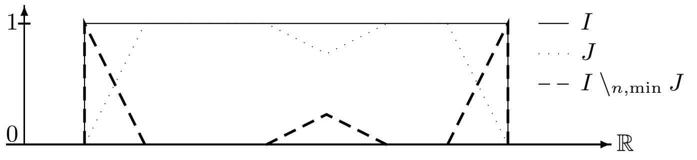
$$(I \setminus_{N,T} J) \overset{\text{def}}{=} I \cap_T N(J)$$

■



Fuzzy Set Difference $I \setminus_{N_{0.5},T_{0.5}} J$

Splitting an interval into two intervals is the worst that can happen for the set difference of two crisp intervals. In the case of fuzzy intervals, the set difference operator can produce arbitrary many disjoint intervals, as the next figure shows.

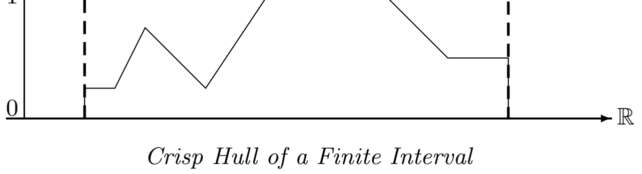


Set Difference Splits into three Components

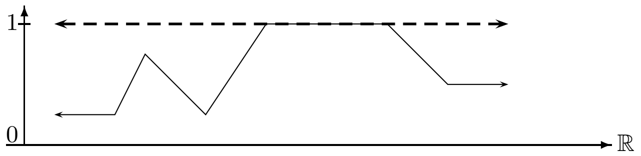
2.2.5 Hull Operators for Fuzzy Intervals

Except for the closed hull of an open interval there is no meaningful notion of a 'hull' for a single crisp time interval. It turns out, however, that there are various *hulls* for fuzzy intervals. We define them in the order of information loss. The first notion of a hull, the *crisp hull* looses most information about the interval, whereas the last notion, the *monotone hull* looses the least information. All these notions of a hull coincide for crisp intervals.

Definition 2.2.27 (Crisp Hull) *For an interval $I \in F_{\mathbb{R}}$ let $crispHull(I)$ be the smallest convex crisp interval containing I.* ■



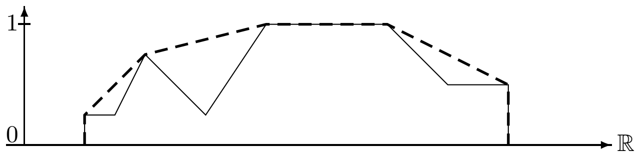
Crisp Hull of a Finite Interval



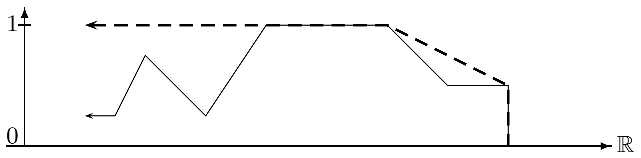
Crisp Hull of an Infinite Interval

Notice that $crispHull(I) \neq S^{\sqcap}(I)$ if I consists of several unconnected components.

Definition 2.2.28 (Convex Hull) *The* convex hull *$convexHull(I)$ of a fuzzy set I is the smallest convex set containing I.* ■



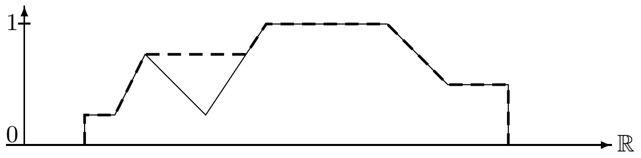
Convex Hull of a Finite Set



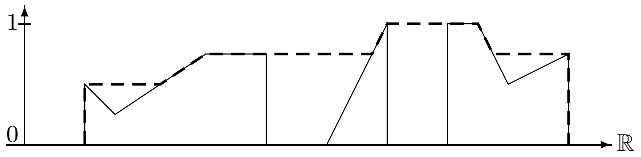
Convex Hull of an Infinite Set

Finally we define the *monotone hull* which looses the least of the structural information about the interval.

Definition 2.2.29 (Monotone Hull) *The* monotone hull *$monotoneHull(I)$ of a fuzzy set I is the smallest* monotone *fuzzy interval containing I.* Monotone *means that from left to right the fuzzy values $monotoneHull(I)(x)$ are rising monotonically to* $\sup(I)$*, and then falling monotonically again.* ■



Monotone Hull



Monotone Hull of a Fuzzy Interval with Three Components

2.2.6 Basic Unary Transformations

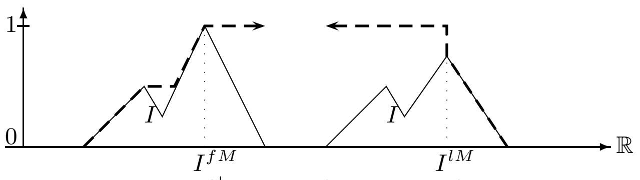
We now introduce a little library of interval operators. They are used in the GeTS specification language [32] as building blocks, for example, to define fuzzy point–interval and fuzzy interval–interval relations [34].

Definition 2.2.30 (Basic Unary Transformations) *Let $I \in F_{\mathbb{R}}$ be a fuzzy interval. We define the following (parameterised) interval operators:*

$$\begin{array}{lll}
\mathit{identity}(I) & \stackrel{\text{def}}{=} & I \\[1ex]
\mathit{extend}^+(I)(x) & \stackrel{\text{def}}{=} & \begin{cases} \sup\{I(y) \mid y \le x\} & \text{if } x \le I^{fM} \\ 1 & \text{otherwise} \end{cases} \\[1ex]
\mathit{extend}^-(I)(x) & \stackrel{\text{def}}{=} & \begin{cases} \sup\{I(y) \mid y \ge x\} & \text{if } x \ge I^{lM} \\ 1 & \text{otherwise} \end{cases} \\[1ex]
\mathit{scaleup}(I)(x) & \stackrel{\text{def}}{=} & \begin{cases} I(x)/\sup(I) & \text{if } \sup(I) \neq 0 \\ 0 & \text{otherwise} \end{cases} \\[1ex]
\mathit{cut}_{x_1,x_2}(I)(x) & \stackrel{\text{def}}{=} & \begin{cases} 0 & \text{if } x < x_1 \text{ or } x \ge x_2 \\ I(x) & \text{otherwise} \end{cases} \\[1ex]
\mathit{cut}_{x_1,+}(I)(x) & \stackrel{\text{def}}{=} & \begin{cases} 0 & \text{if } x < x_1 \\ I(x) & \text{otherwise} \end{cases} \\[1ex]
\mathit{cut}_{x_1,-}(I)(x) & \stackrel{\text{def}}{=} & \begin{cases} 0 & \text{if } x \ge x_1 \\ I(x) & \text{otherwise} \end{cases} \\[1ex]
\mathit{shift}_n(I)(x) & \stackrel{\text{def}}{=} & I(x-n) \\[1ex]
\mathit{times}_a(I)(x) & \stackrel{\text{def}}{=} & \min(1, a \cdot I(x)) \qquad a \ge 0 \\[1ex]
\mathit{exponentiate}_e(I)(x) & \stackrel{\text{def}}{=} & I(x)^e \qquad\qquad e \ge 0 \\[1ex]
\mathit{integrate}^+(I)(x) & \stackrel{\text{def}}{=} & \lim_{a \mapsto \infty} \dfrac{\int_{-a}^{x} I(y)dy}{\int_{-a}^{+a} I(y)dy} \\[1ex]
\mathit{integrate}^-(I)(x) & \stackrel{\text{def}}{=} & \lim_{a \mapsto \infty} \dfrac{\int_{x}^{+a} I(y)dy}{\int_{-a}^{+a} I(y)dy} \\[1ex]
\mathit{negate}_{\mathit{offset}}(I)(x) & \stackrel{\text{def}}{=} & 1 - I(x - \mathit{offset}) \\[1ex]
\mathit{invert}(I)(x) & \stackrel{\text{def}}{=} & \begin{cases} 1 - I(x) & \text{if } I_k^{fM} \le x < I_{k+1}^{lM} \\ & \text{where } I_0, \ldots I_m \text{ are the components of } I \\ 0 & \text{otherwise.} \end{cases}
\end{array}$$

■

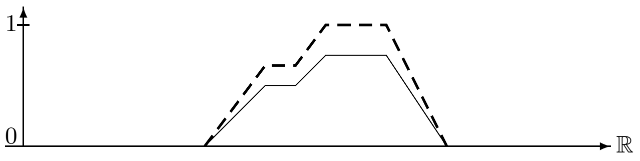
extend
extend$^+(I)$ follows the left part of the monotone hull of the interval until the left maximum I^{lM} is reached and then stays at fuzzy value 1. *extend*$^-(I)$ is the symmetric version of *extend*$^+(I)$.



extend$^+$ and *extend*$^-$

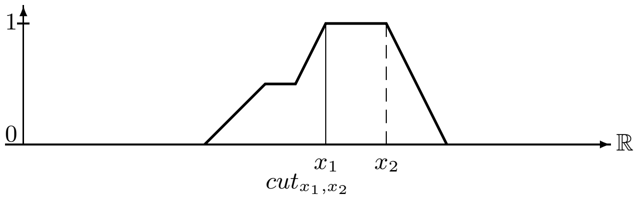
extend$^+(I)$ is useful for implementing a fuzzy 'before'-relation because only the left part of I is relevant for evaluating 'before'. *extend*$^-(I)$, on the other hand, can be used for an 'after'-relation.

scaleup
The *scaleup*-function is different to the *identity* function only if the hight $\sup(I)$ is not 1. In this case it scales the membership function up such that $\sup(scaleup(I)) = 1$.



scaleup

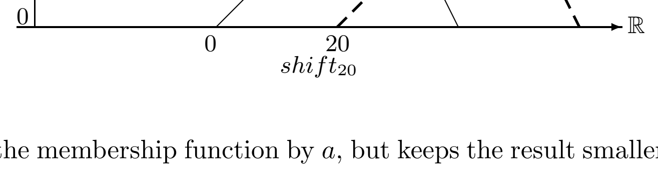
cut
$cut_{x_1,x_2}(I)$ just cuts the piece between x_1 and x_2 out of the interval I. The resulting interval is closed at x_1 and half open at x_2.



cut_{x_1,x_2}

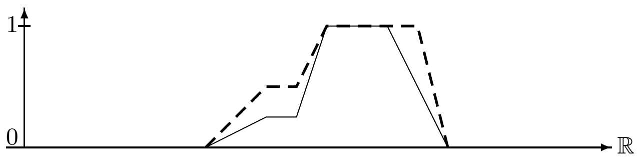
$cut_{x_1,+}(I)$ cuts the part out of I before x_1 whereas $cut_{x_1,-}(I)$ cuts the part out of I after x_1.

shift
$shift_n$ just moves the interval by n time units.



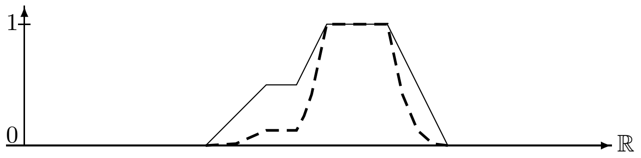
$shift_{20}$

times
$times_a$ multiplies the membership function by a, but keeps the result smaller or equal 1. $times_a$ has no effect on crisp intervals.



$times_2$

exponentiate
$exponentiate_e$ takes the membership function to the exponent e. It can be used to damp increases or decreases. $exponentiate_e$ has also no effect on crisp intervals. $exponentiate_e$ is non-linear in the sense that straight lines are turned into curved lines.



$exponentiate_3$

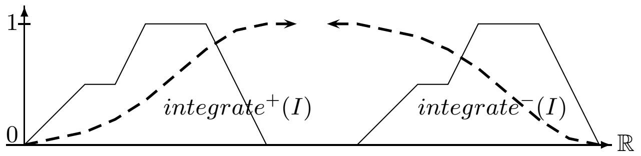
integrate
This operator integrates over the membership function and normalises the integral to values ≤ 1. The two integration operators $integrate^+$ and $integrate^-$ can be simplified for finite fuzzy time intervals.

Proposition 2.2.31 (Integration for Finite Intervals) *If the fuzzy interval I is finite then*

$$integrate^+(I)(x) = \frac{\int_{-\infty}^{x} I(y)dy}{|I|} \qquad and \qquad integrate^-(I)(x) = \frac{\int_{x}^{+\infty} I(y)dy}{|I|}$$

The proofs are straightforward. ■

Example for $integrate^+$ and $integrate^-$:



$integrate^+$ and $integrate^-$

The integration operator for infinite intervals I with finite kernel turns the interval into a constant function which does no longer depend on the finite part of I.

Proposition 2.2.32 (Integration for Intervals with Finite Kernel) *If the infinite fuzzy interval I has a finite kernel with $I_1 \overset{\text{def}}{=} I(-\infty)$ and $I_2 \overset{\text{def}}{=} I(+\infty)$ then*

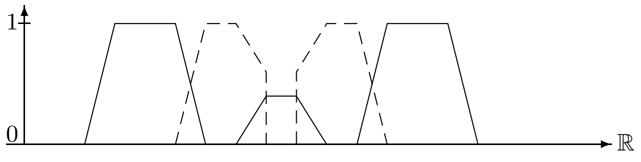
$$integrate^+(I)(x) = \frac{I_1}{I_1 + I_2} \qquad and \qquad integrate^-(I)(x) = \frac{I_2}{I_1 + I_2}$$

Proof:

$$\begin{aligned}
integrate^{+}(I)(x) &= \lim_{a\mapsto\infty}\frac{\int_{-a}^{x} I(y)dy}{\int_{-a}^{+a} I(y)dy}\\
&= \lim_{a\mapsto\infty}\frac{|I|_{-a}^{I^{fK}}+|I|_{I^{fK}}^{x}}{|I|_{-a}^{I^{fK}}+|I|_{I^{fK}}^{I^{lK}}+|I|_{I^{lK}}^{a}}\\
&= \lim_{a\mapsto\infty}\frac{|I|_{-a}^{I^{fK}}}{|I|_{-a}^{I^{fK}}+|I|_{I^{lK}}^{a}}\\
&= \lim_{a\mapsto\infty}\frac{(I^{fK}+a)\cdot I_1}{(I^{fK}+a)\cdot I_1+(a-I^{lK})\cdot I_2}\\
&= \lim_{a\mapsto\infty}\frac{a\cdot I_1}{a\cdot I_1+a\cdot I_2}\\
&= \frac{I_1}{I_1+I_2}\\
integrate^{-}(I)(x) &= \lim_{a\mapsto\infty}\frac{\int_{x}^{+a} I(y)dy}{\int_{-a}^{+a} I(y)dy}\\
&= \lim_{a\mapsto\infty}\frac{|I|_{x}^{I^{lK}}+|I|_{I^{lK}}^{a}}{|I|_{-a}^{I^{fK}}+|I|_{I^{fK}}^{I^{lK}}+|I|_{I^{lK}}^{a}}\\
&= \lim_{a\mapsto\infty}\frac{|I|_{I^{lK}}^{a}}{|I|_{-a}^{I^{fK}}+|I|_{I^{lK}}^{a}}\\
&= \lim_{a\mapsto\infty}\frac{(a-I^{lK})\cdot I_2}{(I^{fK}+a)\cdot I_1+(a-I^{lK})\cdot I_2}\\
&= \lim_{a\mapsto\infty}\frac{a\cdot I_2}{a\cdot I_1+a\cdot I_2}\\
&= \frac{I_2}{I_1+I_2}
\end{aligned}$$

■

invert
The *invert* function is almost like the standard negation function, except that *invert*(I) is nonzero only in the gaps between the components of I. The interval I in the next picture consists of three components. The maximal fuzzy value of the middle component is not 1. Nevertheless *invert*(I) drops down to 0 between the first and last maximum of the middle component. *invert* is needed for an *in_the_gap* operator.



invert

Fuzzification
Fuzzy time intervals could be defined by specifying the shape of the membership function in some way. This is in general very inconvenient. Therefore FuTI provides an alternative. The idea is to take a crisp interval and to 'fuzzify' the front and back end in a certain way. For example, one may specify 'early afternoon' by taking the interval between 1 and 6 pm and imposing, for example, a linear or a Gaussian shape increase from 1 to 2 pm, and a linear or a Gaussian shape decrease from 4 to 6 pm. Technically this means multiplying a linear or Gaussian function with the membership values.

The fuzzification functions can be defined with absolute coordinates and with relative coordinates. We define the absolute version first.

Definition 2.2.33 (Linear Fuzzification Function) *Let $I \in F_{\mathbb{R}}$, x_1, x_2 and offset be x-coordinates.*

We define the 'front' linear fuzzification function with zero offset first:

$$FALf_{x_1,x_2,0}(I)(x) \overset{\text{def}}{=} \begin{cases} 0 & \textit{if } x < x_1 \\ I(x) & \textit{if } x \geq x_2 \\ I(x)\frac{x-x_1}{x_2-x_1} & \textit{otherwise} \end{cases}$$

If the offset is nonzero we have

$$FALf_{x_1,x_2,\textit{offset}}(I)(x) \overset{\text{def}}{=} \begin{cases} FALf_{x_1,x_2,0}(x+\textit{offset}) & \textit{if } x < x_2 - \textit{offset} \\ FALf_{x_1,x_2,0}(x_2) & \textit{if } x_2 - \textit{offset} \leq x < x_2 \\ I(x) & \textit{otherwise} \end{cases}$$

The 'back' linear fuzzification function is:

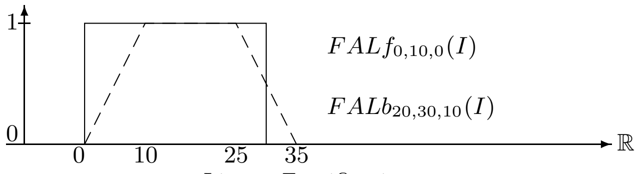
$$FALb_{x_1,x_2,0}(I)(x) \overset{\text{def}}{=} \begin{cases} 0 & \textit{if } x \geq x_2 \\ I(x) & \textit{if } x < x_1 \\ I(x)\frac{x_2-x}{x_2-x_1} & \textit{otherwise} \end{cases}$$

If the offset is nonzero we have

$$FALb_{x_1,x_2,\textit{offset}}(I)(x) \overset{\text{def}}{=} \begin{cases} FALb_{x_1,x_2,0}(x-\textit{offset}) & \textit{if } x \geq x_1 + \textit{offset} \\ FALb_{x_1,x_2,0}(x_2) & \textit{if } x_1 \leq x \leq x_1 + \textit{offset} \\ I(x) & \textit{otherwise} \end{cases}$$

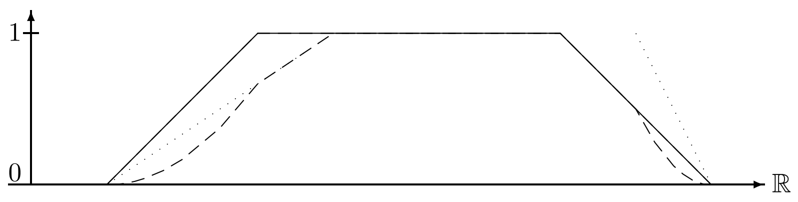
■

In the picture below we fuzzify a crisp interval with a linear increase from $0 - 10$, and a linear decrease from $20 - 30$, which is shifted by an offset of 10.



Linear Fuzzification:

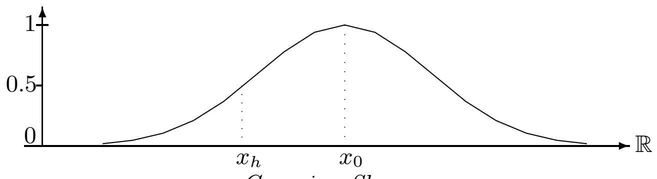
The next example shows the linear fuzzification of an already fuzzy interval. The dotted lines show the linear increase and decrease. The dashed line is the result of the fuzzification operator. Since the two polygons are multiplied, we get quadratic curves.



Linear Fuzzification of an Already Fuzzy Interval

Gaussian Fuzzification

Besides linear fuzzification, FuTI offers the fuzzification with a Gaussian shape. The Gaussian function is $e^{-(\frac{x-x_0}{\sigma})^2}$. x_0 is the symmetry point and σ determines the increase and decrease.



Gaussian Shape

The Gaussian fuzzification function is determined by the parameters x_0 and x_h. x_h is the x-coordinate where $e^{-((x_h-x_0)/\sigma)^2} = 0.5$. This condition determines $\sigma = \sqrt{(-1/ln(0.5))}\cdot(x_h-x_0)$.

Since the Gauss function does not become 0, we must cut it off at some x-coordinate. The heuristic is to cut it off at a distance $3(x_0 - x_h)$ from x_0.

Definition 2.2.34 (Gaussian Fuzzification Function) *Let $I \in F_{\mathbb{R}}$, x_h, x_0 and offset be x-coordinates. Let $\sigma \overset{\text{def}}{=} \sqrt{(-1/ln(0.5))} \cdot (x_h - x_0)$.*

We define the 'front' Gaussian fuzzification function with zero 'offset' first:

$$FAGf_{x_h,x_0,0}(I)(x) \overset{\text{def}}{=} \begin{cases} 0 & \textit{if } x < 3x_h - 2x_0 \\ I(x) & \textit{if } x \geq x_0 \\ I(x)e^{-((x-x_0)/\sigma)^2} & \textit{otherwise} \end{cases}$$

If the offset is nonzero we have

$$FAGf_{x_h,x_0,\mathit{offset}}(I)(x) \overset{\text{def}}{=} \begin{cases} FAGf_{x_h,x_0,0}(x + \mathit{offset}) & \textit{if } x < x_0 - \mathit{offset} \\ FAGf_{x_h,x_0,0}(x_0) & \textit{if } x_0 - \mathit{offset} \leq x < x_0 \\ I(x) & \textit{otherwise} \end{cases}$$

The 'back' Gaussian fuzzification function is:

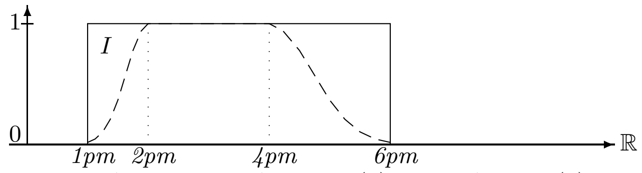
$$FAGb_{x_h,x_0,0}(I)(x) \overset{\text{def}}{=} \begin{cases} 0 & \textit{if } x > 3x_h - 2x_0 \\ I(x) & \textit{if } x < x_0 \\ I(x)e^{-((x-x_0)/\sigma)^2} & \textit{otherwise} \end{cases}$$

If the offset is nonzero we have

$$FAGb_{x_h,x_0,\mathit{offset}}(I)(x) \overset{\text{def}}{=} \begin{cases} FAGb_{x_h,x_0,0}(x - \mathit{offset}) & \textit{if } x \geq x_0 + \mathit{offset} \\ FAGb_{x_h,x_0,0}(x_2) & \textit{if } x_0 \leq x \leq x_0 + \mathit{offset} \\ I(x) & \textit{otherwise} \end{cases}$$

■

Example 2.2.35 *We fuzzify 'early afternoon' by taking the interval between 1pm and 6pm, imposing a Gaussian rise between 1pm and 2pm and a Gaussian decrease between 4 and 6pm.*



■

Early Afternoon: $FAGf_{1.5,2,0}(I)$ and $FAGb_{5,4,0}(I)$

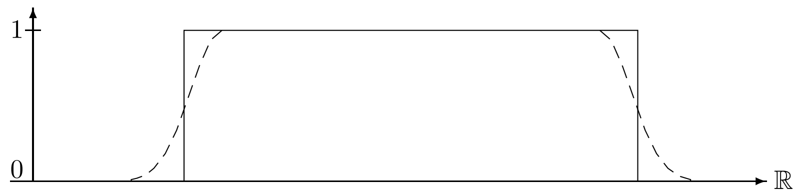
Fuzzification functions with absolute coordinates are not that useful because usually one does not know the coordinates in advance. Therefore FuTI also provides fuzzification functions where the parameters are percentage values. $FRLf_{10,5}$, for example, means linear fuzzification where the linear increase is 10% of the kernel size and the offset is 5% of the kernel size. $FRGf_{10,0}$ means a Gaussian increase where x_0 is 10% of the kernel size past I^{fK}, x_h is 1/2 the distance between I^{fK} and x_0, and the offset is such that x_h coincides with I^{fK}.

Definition 2.2.36 (Fuzzification with Relative Coordinates) *For an interval I, percentage numbers r and o between 0 and 100 we define the relative fuzzification functions.*
Let $d = (I^{lK} - I^{fK})/100$.

$$\begin{array}{lcl}
FRLf_{r,o}(I) & \overset{\text{def}}{=} & FALf_{I^{fK},I^{fK}+d\cdot r,I^{fK}-d\cdot o}(I) \\
FRLb_{r,o}(I) & \overset{\text{def}}{=} & FALb_{I^{lK}-d\cdot r,I^{lK},I^{lK}+d\cdot o}(I) \\
FRL_{r,o}(I) & \overset{\text{def}}{=} & FALb_{I^{lK}-d\cdot r,I^{lK},I^{lK}+d\cdot o}(FALf_{I^{fK},I^{fK}+d\cdot r,I^{fK}-d\cdot o}(I)) \\
FRGf_{r}(I) & \overset{\text{def}}{=} & FAGf_{I^{fK}+d\cdot r,I^{fK}+1/2\cdot d\cdot r,2/3\cdot d\cdot r}(I) \\
FRGb_{r}(I) & \overset{\text{def}}{=} & FAGb_{I^{lK}-d\cdot r,I^{lK}-1/2\cdot d\cdot r,2/3\cdot d\cdot r}(I) \\
FRG_{r}(I) & \overset{\text{def}}{=} & FAGb_{I^{lK}-d\cdot r,I^{lK}-1/2\cdot d\cdot r,2/3\cdot d\cdot r}(FAGf_{I^{fK}+d\cdot r,I^{fK}+1/2\cdot d\cdot r,2/3\cdot d\cdot r}(I))
\end{array}$$

■

The functions FRL and FRG fuzzify an interval at both sides. A simple composition of $FRLf$ and $FRLb$, for example, yields an un-symmetric result because the fuzzification at one end first changes the kernel size. The relative fuzzification of the other side of the changed literal therefore uses the data of the changed interval for computing x_h and x_0. FRL and FRG avoid this by computing the absolute coordinates first and using them for both sides.



Relative Gaussian Fuzzification FRG_{20}

2.3 Data Structures and Algorithms

The main data structures and algorithms of the FuTI–module are presented in this section. The actual implementation contains a few more functions. Since their implementation is more or less straight forward, they are not mentioned explicitely here. There are four basic datatypes: time points, fuzzy values, fuzzy temporal intervals and y-functions.

Time Points
The time points are points on the ℝ-axis. Arbitrary real numbers cannot be represented on computers. The choice is therefore between floating point numbers and integers as representation of time points. The range of floating point numbers is much higher than the range of integers. Unfortunately, algorithms operating on floating point numbers are prone to uncontrollable rounding errors. Another argument for using integers instead of floating point numbers is that the real time measurements on earth give you always integers. The very definition of exact time measurement already uses integers: in 1967 one second was defined as 9.192.631.770 cycles of the light emitted when an electron jumps between the the two lowest hyperfine levels of the Caesium 133 atom. Thus, the most precise time measurement available at all depends on counting integers (cycles of light).

Therefore the FuTI–module *represents time with integer coordinates.* There is no assumption about the meaning of these integers. They may be years, seconds, picoseconds or even cycles of the Caesium 133 light.

Fuzzy Values
Fuzzy values are usually real numbers between 0 and 1. A first choice would therefore be to use floating point numbers for the fuzzy values. Again, floating point numbers are prone to rounding errors. Moreover, computation with floating point numbers is more expensive than computation with integers. Therefore FuTI uses again integers instead of floating point numbers. This means of course that one cannot represent the fuzzy value 1 as the integer 1. We could then use just 0 and 1 and no other fuzzy value. Instead one better represents the fuzzy value 1 as a suitable unsigned integer of a certain bit size. Since fuzzy values are estimates only anyway, 16 bit unsigned integer (unsigned short int in C) are precise enough for fuzzy values.

Definition 2.3.1 (Largest Fuzzy Value) *Let $\top$ be the maximal fuzzy value in the implementation.* ■

To make the examples more easy to understand, we use $\top = 1000$ in this paper. $\top$ is a compiler option in the actual implementation and can be changed easily.

Fuzzy Time Intervals
Fuzzy intervals are usually implemented by a representation of their membership functions. Arbitrary membership functions are almost impossible to represent precisely on a computer. A natural choice for realizing approximated fuzzy time intervals over integer time and integer fuzzy values is the representation with *envelope polygons* over integer coordinates. This has a number of advantages: the representation is compact and can nevertheless approximate the membership functions very well; simple structures, like crisp intervals, have a simple representation; we can use ideas and algorithms from Computational Geometry [36, 19]; there are very efficient algorithms for most of the problems, and it is clear where rounding errors can occur, and where not.

Coordinates and Integer Datatypes
The implemented fuzzy intervals are independent of their interpretation as fuzzy time intervals. Therefore we shall speak of the x-axis instead of the time axis and of the y-axis instead of the fuzzy value axis.

The Used Coordinate System

Definition 2.3.2 (x-Integers and y-Integers) *FuTI uses integers of different size for the x-coordinates and the y-coordinates. Therefore we shall speak of the x-integers and of the y-integers. The default for x-integers is 64 bit long long integers, and the default for y-integers is 16 bit short integers. The module has also been tested with multiple-precision x-integers.* ■

Notation for Algorithms
We shall write most algorithms in a functional notation which is as mathematical as possible,

but still concrete enough that they can be implemented straight away. It turned out that the object oriented paradigm is not only very good for getting modularised and easy to understand implementations, but it also makes the mathematical notation clearer. Therefore we shall use the notation $o.v$ and $o.m(p_1, \ldots, p_n)$ where o is an object, v is an instance variable, and m is a method (function) with arguments $p_1, \ldots, p_n$.

The expression

$$a \stackrel{\text{def}}{=} \begin{cases} s_1 & \text{if } \varphi_1 \\ s_2 & \text{if } \varphi_2 \\ \ldots & \ldots \\ s_n & \text{otherwise} \end{cases}$$

is a case analysis. It means:
$a \stackrel{\text{def}}{=} s_1$ if φ_1 is true
$a \stackrel{\text{def}}{=} s_2$ if φ_1 is false and φ_2 is true
...
$a \stackrel{\text{def}}{=} s_n$ if $\varphi_1, \ldots, \varphi_{n-1}$ are all false.

The notation $\Sigma_{n=0}^{m} s(n)$ is well known in mathematics. In the same style we define a notation $V_{n=0}^{m} s(n)$. The V operator causes the values $s(n)$ to be collected in a list. For example,

$$V_{n=0}^{20} \begin{cases} (n) & \text{if } n \text{ is a prime number} \\ () & \text{otherwise} \end{cases}$$

yields the list (1,3,5,7,11,13,17,19).

We may also use the keyword *break* to stop the V-loop. For example,

$$V_{n>0} \begin{cases} \mathit{break} & \text{if } n > 20 \\ (n) & \text{if } n \text{ is a prime number} \\ () & \text{otherwise} \end{cases}$$

yields the same list (1,3,5,7,11,13,17,19).

Sometimes it is necessary to include a value in a list and then stop the loop. We specify this with an expression 's and *break*'.

$$V_{n>0} \begin{cases} (n) \text{ and } \mathit{break} & \text{if } n > 20 \\ (n) & \text{if } n \text{ is a prime number} \\ () & \text{otherwise} \end{cases}$$

yields the list (1,3,5,7,11,13,17,19,21).

Partial Functions and Error Handling
Most of the functions defined in this chapter are partial functions. Therefore the preconditions the arguments must meet when these functions are called need to be stated very clearly. This means for an implementation that the functions should only be called when the preconditions are guaranteed. An error handling mechanism treats the cases where the preconditions are not met.

Special Functions
We use the following functions:

$roundX(a)$ rounds the floating point number a to the closest x-integer (time value).

$roundY(a)$ rounds the floating point number a to the closest y-integer (fuzzy value).

The two functions are almost identical. The only difference is the bit length of the resulting integer values.

2.3.1 Points

We need 2-dimensional points with coordinates (x, y) as the representation of points on the envelope polygon. The x-coordinate is the time coordinate and the y-coordinate is the fuzzy value coordinate. x-coordinates are represented with x-integers and y-coordinates are represented with y-integers (Def. 2.3.2).

Notation
If $p = (x, y)$ is a point then $p.x$ denotes the x-coordinate (time coordinate) of p and $p.y$ denotes the y-coordinate (fuzzy coordinate) of p.

Collinearity
The collinearity check for three points p_1, p_2 and p_3 is a standard method from Computational Geometry [36]. The doubled area of the triangle p_1, p_2 and p_3 is computed. With integer coordinates this can be done without any error at all. If the doubled area is 0 then the three points are collinear.

Definition 2.3.3 (Collinear) *The method $p_1.collinear(p_2, p_3)$ returns true if the three points p_1, p_2 and p_3 are collinear.* ∎

Left turn
Another important operator is the 'left turn test'.

Definition 2.3.4 (Left Turn) *The method $p_1.leftturn(p_2, p_3)$ returns true if the three points p_1, p_2 and p_3 make a left turn.* ∎

The $leftturn$ method computes the doubled area of the triangle p_1, p_2 and p_3 and checks its sign. Left turns and right turns yield opposite signs.

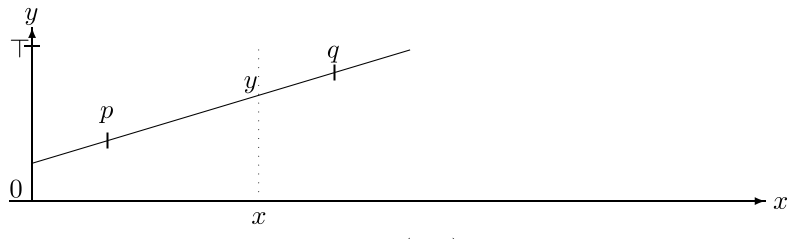
Intersection
Testing whether line segments intersect and computing the intersection point are also standard methods from Computational Geometry.

Definition 2.3.5 (Intersects and IntersectsProper) *The method $p_1.intersects(p_2, q_1, q_2)$ returns true if the line segment (p_1, p_2) intersects or touches the line segment (q_1, q_2).*

The method $p_1.intersectsProper(p_2, q_1, q_2)$ returns true if the line segment (p_1, p_2) intersects properly, and not only touches the line segment (q_1, q_2). ∎

Definition 2.3.6 (Intersection) *The method $p_1.intersection(p_2, q_1, q_2)$ returns the rounded x-coordinate of the intersection point of the two intersecting line segments (p_1, p_2) and (q_1, q_2).* ∎

lineY
The function $p.lineY(q,x)$ considers the line crossing the points p and q, and computes for a given x-value the corresponding y value at the line.



$p.lineY(q,x)$

Definition 2.3.7 (lineY) *Let p and q be the two points which define a line, and let x be an x-coordinate.*

$$p.lineY(q,x) \overset{\text{def}}{=} \begin{cases} undefined & \text{if } p.x = q.x \\ p.y + \frac{(q.y-p.y)\cdot(x-p.x)}{q.x-p.x} & \text{otherwise} \end{cases}$$

The result is floating point number. ■

lineX
This method computes for a line and a y-value the corresponding x-value.

Definition 2.3.8 (lineX) *Let p and q be the two points which define a line, and let y be a y-coordinate.*

$$p.lineX(q,y) \overset{\text{def}}{=} \begin{cases} undefined & \text{if } p.y = q.y \\ p.x + roundX(\frac{(q.x-p.x)\cdot(y-p.y)}{q.y-p.y}) & \text{otherwise} \end{cases}$$

The result is an x-coordinate. ■

Area
We provide two methods for computing the area between a line and the x-axis. The first function $p.area2(q)$ computes for two points p and q twice the area below the line segment between p and q. When p and q are points with integer coordinates then twice the area yields also an integer, and no rounding is necessary.

The second method $p.area2(q, x_1, x_2)$ computes twice the area between x_1 and x_2 below the line segment between p and q.

Definition 2.3.9 (area2) *Let p and q be the two points which define a line, let x_1 and x_2 x-coordinates.*

$$p.area2(q) \overset{\text{def}}{=} (q.x - p.x)\cdot(q.y + p.y)$$

The result is an x-integer.

$$p.area2(q,x_1,x_2) \overset{\text{def}}{=} \begin{cases} undefined & \text{if } p.x = q.x \text{ and } p.x \neq x \\ 0 & \text{if } p.x = q.x = x \\ (x_2 - x_1)\cdot(p.lineY(q,x_2) - p.lineY(q,x_1)) & \text{otherwise} \end{cases}$$

The result is a floating point number. ■

The next method, $p.area2X(q,a)$ computes for two points p and q and for a doubled area a the x-coordinate x such that twice the area below the line segment between p and q from $p.x$ till x is a. The function is undefined if the line is vertical, or the line is just the coordinate axis and $a > 0$, or the slope of the line is negative and there is not enough area available between $p.x$ and the point where the line crosses the coordinate axis.

Definition 2.3.10 (area2X) *Let p and q be the two points which define a line. Let $a \geq 0$ be an integer or floating point number.*

$$p.area2X(q,a) \stackrel{\text{def}}{=} \begin{cases} undefined & if\ p.x = q.x \\ & or\ p.y = q.y = 0\ and\ a > 0 \\ & or\ p.y^2 < -slope \cdot a \\ p.x & if\ p.y = q.y = 0\ and\ a = 0 \\ p.x + roundX(\frac{a}{2p.y}) & if\ p.y = q.y \\ p.x + roundX(\frac{\sqrt{p.y^2+slope\cdot a}-p.y}{slope}) & otherwise \\ \quad where\ slope \stackrel{\text{def}}{=} \frac{q.y-p.y}{q.x-p.x} & \end{cases}$$

The result is a rounded x-integer. ∎

Proposition 2.3.11 (Soundness of $area2X$) *Let p and q be two points and a a doubled area (non-negative number). Then $p.area2X(q,a)$ returns the (rounded) x-coordinate x such that the doubled area below the line crossing p and q and between $p.x$ and x equals a.*

Proof: The doubled area below the line crossing p and q and between $p.x$ and x is
$(x - p.x) \cdot (p.y + (p.y + slope \cdot (x - p.x))) = a$
where $slope = \frac{q.y-p.y}{q.x-p.x}$

Case 1: $q.x - p.x = 0$, i.e. $p.x = q.x$.
The equation is not solvable in this case.

Case 2: $slope = 0$, i.e. $p.y = q.y$:
Case 2a: $p.y = 0$: the equation is only solvable for $a = 0$, in which case $p.x$ is a solution.
Case 2b: $p.y > 0$: The equation simplifies in this case to
$(x - p.x) \cdot 2p.y+ = a$ with solution
$x = p.x + \frac{a}{2p.y}$.

Case 3: $slope \neq 0$:
The equation is normalised to
$slope \cdot (x - p.x)^2 + 2p.y(x - p.x) - a = 0$ with solution
$(x - p.x) = \frac{-2p.y \pm \sqrt{4p.y^2+4slope\cdot a}}{2slope}$
$x = p.x + \frac{-p.y+\sqrt{p.y^2+slope\cdot a}}{slope}$
The $-\sqrt{\ldots}$-case yields a point left of $p.x$, which is not what we want. The square root has a real number solution only if $p.y^2 + slope \cdot a \geq 0$. Otherwise the function is undefined. ∎

Integration
Some interval–interval relations are defined as an integral over two multiplied polygons (Section 2.3.3). A building block for the integration algorithm is a method which integrates the product of two lines.

Definition 2.3.12 (Integration of Multiplied Lines) *Let p_1, p_2 and q_1, q_2 be the two pairs of points which define two lines. Let x_1 and x_2 be two x-coordinates.*

$$p_1.integrate(p_2, q_1, q_2, x_1, x_2) \overset{\text{def}}{=} \begin{cases} undefined & \text{if } (p_1.x = p_2.x \text{ or } q_1.x = q_2.x) \text{ and } x_1 \neq x_2 \\ 0 & \text{if } x_1 = x_2 \\ a \cdot b \cdot (x_2 - x_1) + (m_2 a + m_1 b) \cdot (x_2^2 - x_1^2)/2 + m_1 \cdot m_2 \cdot (x_2^3 - x_1^3)/3 & \text{otherwise} \end{cases}$$

where $a \overset{\text{def}}{=} p_1.y - m_1 p_1.x, \quad b \overset{\text{def}}{=} q_1.y - m_2 q_2.x,$

$$m_1 \overset{\text{def}}{=} \frac{p_2.y - p_1.y}{p_2.x - p_1.x} \text{ and } m_2 \overset{\text{def}}{=} \frac{q_2.y - q_1.y}{q_2.x - q_1.x}.$$

The result is a floating point number. ∎

Proposition 2.3.13 (Soundness of Integration of Multiplied Lines) *Let p_1, p_2 and q_1, q_2 be the two pairs of points which define two lines. Let x_1 and x_2 be two x-coordinates. Then*

$$p_1.integrate(p_2, q_1, q_2, x, y) = \int_{x_1}^{x_2} l_1(x) \cdot l_2(x)\ dx$$

where l_1 is the line crossing p_1 and p_2 and l_2 is the line crossing q_1 and q_2.

Proof: Let $l_1(x) \overset{\text{def}}{=} p_1.y + m_1(x - p_1.x)$ and $l_2(x) \overset{\text{def}}{=} q_1.y + m_2(x - q_1.x)$ where $m_1 \overset{\text{def}}{=} \frac{p_2.y - p_1.y}{p_2.x - p_1.x}$ and $m_2 \overset{\text{def}}{=} \frac{q_2.y - q_1.y}{q_2.x - q_1.x}$.

$$\begin{aligned}
&\int_{x_1}^{x_2} l_1(x) \cdot l_2(x)\ dx \\
&= \int_{x_1}^{x_2} (p_1.y + m_1(x - p_1.x))(q_1.y + m_2(x - q_1.x))\ dx \\
&= [(p_1.y - m_1 p_1.x)(q_1.y - m_2 q_2.x) + (m_2(p_1.y - m_1 p_1.x) + m_1(q_1.y - m_2 q_2.x))x + m_1 m_2 x^2]_{x_1}^{x_2} \\
&= [ab + (m_2 a + m_1 b)x + m_1 m_2 x^2]_{x_1}^{x_2} \\
&= ab(x_2 - x_1) + (m_2 a + m_1 b)(x_2^2 - x_1^2)/2 + m_1 m_2 (x_2^3 - x_1^3)/3
\end{aligned}$$

where $a \overset{\text{def}}{=} p_1.y - m_1 p_1.x$ and $b \overset{\text{def}}{=} q_1.y - m_2 q_2.x$. ∎

2.3.2 Fuzzy Time Intervals

In this section we introduce a concrete representation of fuzzy time intervals and present the algorithms implemented in FuTI.

Definition 2.3.14 (Infinity) *We use $+\infty$ and $-\infty$ with the same meaning as before. However, since infinity cannot be represented properly on a computer, $+\infty$ stands in fact for a sufficiently large positive representable x-integer, and $-\infty$ stands for a sufficiently large negative representable x-integer. If the bit size of the integers is fixed, these can be the largest representable integers at all. For multiple-precision integers one can choose an arbitrary very large number.* ∎

The finite representation of $+\infty$ and $-\infty$ could in principle cause errors if the time values become extremely large. Therefore one has to check in the application how large the numbers could become and then choose a large enough x-integer datatype.

2.3.2.1 Representation and Construction

Fuzzy intervals are represented by their *envelope polygons.* These polygons represent the membership functions.

Definition 2.3.15 (Envelope Polygon) *The* envelope polygon *I of a fuzzy time interval is a finite sequence of points $p_0, \ldots, p_n$ such that $p_i.x \le p_{i+1}.x$ holds for all i.*

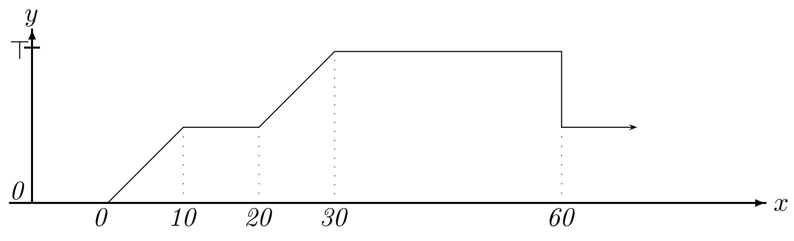
The envelope polygons in FuTI are constructed that there are no redundant points. That means in particular that there are no collinear triples (p_i, p_{i+1}, p_{i+2}) of points.

We usually identify the envelope polygon with the fuzzy interval itself. ■

Example 2.3.16 (Envelope Polygon) *The picture below shows the envelope polygon*

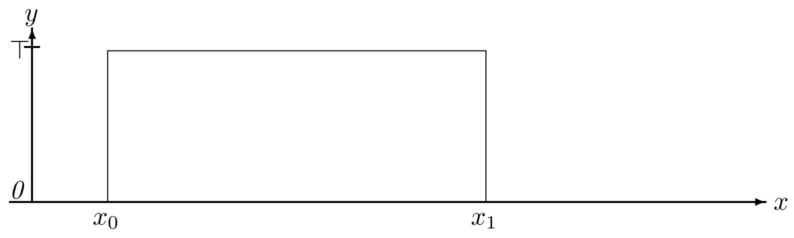
$$I = (0,0)(10,500)(20,500)(30,1000)(60,1000)(60,500).$$

Since $p_5.y = 500 > 0$ it represents a positive infinite fuzzy interval.



The Envelope Polygon

Example 2.3.17 (Crisp Intervals) *The representation of finite crisp intervals consists always of four points: $I = ((x_0, 0)(x_0, \top)(x_1, \top)(x_1, 0)$.*



Finite Crisp Interval

The envelope polygon representation of fuzzy intervals leaves it open whether they are open or closed intervals. This decision is left to the membership function (see Remark 2.3.24 below).

Infinite crisp intervals can of course also be represented. For example, $[10, +\infty[$ can be represented by $(10,0)(10,\top)$. $[-\infty, 10[$ can be represented by $(10,\top)(10,0)$. ■

An envelope polygon is constructed from the empty list of points by adding new points to the back of the list. The *push_back* method defined below ensures that the condition $p_i.x \le p_{i+1}.x$ holds and that collinear triples of points are avoided.

Definition 2.3.18 (push_back and pop_back) *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon and p a new point.*

$$I.push\_back(p) \overset{\text{def}}{=} \begin{cases} undefined & \text{if } I \neq () \text{ and } p.x < p_n.x \\ (p) & \text{if } I = () \text{ or } I = (p_0) \text{ and } p.y = p_0.y \\ (p_0, p) & \text{if } I = (p_0) \\ (p_1, \ldots, p_{n-1}, p) & \text{if } p.collinear(p_{n-1}, p_n) = true \text{ (Def. 2.3.3)} \\ (p_1, \ldots, p_n, p) & \text{otherwise} \end{cases}$$

$I.pop\_back()$ removes the last element. ■

The *push_back* method alone does not guarantee that there are no redundant points in an envelope polygon. The method $I.close()$ defined next removes all remaining redundancies. It is automatically called before the other algorithms use the envelope polygon.

Definition 2.3.19 (Close) *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon.*

$$I.close() \overset{\text{def}}{=} \begin{cases} () & \text{if } I = ((x, 0)) \\ (p_1, \ldots, p_n).close() & \text{if } p_0.y = p_1.y \\ (p_0, \ldots, p_{n-1}) & \text{if } p_{n-1}.y = p_n.y \\ I & \text{otherwise} \end{cases}$$

■

The method *index* defined below can be used to locate for a given x-coordinate x and an envelope polygon I the line segment which is above x. $indexMax(true)$ locates the index of the leftmost polygon point with maximum y-value (I^{fM}), whereas $indexMax(false)$ locates the index of I^{lM}.

Definition 2.3.20 (index and indexMax) *For an envelope polygon $I = (p_0, \ldots, p_n)$ let*

$$I.index(x) \overset{\text{def}}{=} \begin{cases} -1 & \text{if } I = () \text{ or } x < p_0.x \\ \max\{k \leq n \mid x_k \leq x\} & \text{otherwise} \end{cases}$$

be the index of the rightmost polygon point that is left of x. The index is actually obtained with binary search in $O(\log_2(n))$ time.

$$I.indexMax(front) \overset{\text{def}}{=}$$
$$\begin{cases} -1 & \text{if } I = () \\ \min\{i \geq 0 \mid p_i.y = \top \text{ or } \forall j : 0 \leq j < i : \ p_j.y < p_i.y\} & \text{if } front = true \\ \max\{i \leq n \mid p_i.y = \top \text{ or } \forall j : i < j \leq n : \ p_j.y < p_i.y\} & \text{if } front = false \end{cases}$$

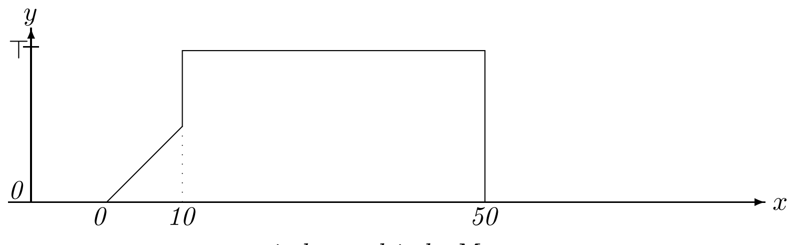
■

indexMax requires linear search. Fortunately the search can be stopped as soon as a point p_i is reached with $p_i.y = \top$. Therefore for the important case of crisp polygons, the search stops always at the second point.

Example 2.3.21 (index and indexMax) *For the envelope polygon*

$$I = \underbrace{(0,0)}_{p_0}\underbrace{(10,500)}_{p_1}\underbrace{(10,1000)}_{p_2}\underbrace{(50,1000)}_{p_3}\underbrace{(50,0)}_{p_4}$$

we have
$I.index(0) = 0$, $I.index(9) = 0$, $I.index(10) = 2$, $I.index(11) = 2$, $I.index(50) = 4$,
$I.indexMax(true) = 2$, $I.indexMax(false) = 3$.



index and indexMax

■

The envelope polygon contains only the vertices of a piecewise linear membership function. Therefore we need a *member* method which interpolates for a given x the corresponding y-value of the membership function.

Definition 2.3.22 (Member Function) *Given a fuzzy interval (envelope polygon)* $I = (p_0, \ldots, p_n)$ *the* Member function *is defined:*

$$I.member(x) \overset{\text{def}}{=} \begin{cases} 0 & \textit{if } I = () \\ p_0.y & \textit{if } x < p_0.x \\ p_n.y & \textit{if } x \geq p_n.x \\ p_i.y & \textit{if } x = p_i.x \\ p_i.lineY(p_{i+1}, x) & \textit{otherwise} \\ \quad \textit{where } i = I.index(x) & \end{cases}$$

The result is converted to a floating point number, if necessary.

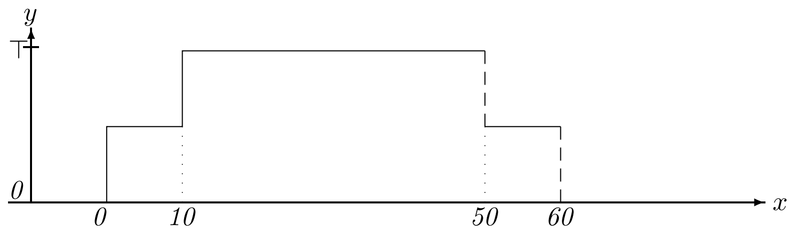
The usual membership function (Def. 2.2.1) is then $I(x) = I.member(I, x)/\top$ ■

Remark 2.3.23 (Extrapolation and Infinite Intervals) *The member method extrapolates the membership function to x-coordinates below* $p_0.x$ *and above* $p_n.x$*. The y-value for x-coordinates below* $p_0.x$ *is constant* $p_0.y$*. The y-value for x-coordinates above* $p_n.x$ *is constant* $p_n.y$*. Therefore envelope polygons always represent fuzzy intervals* with finite kernel *(Def. 2.2.4).* ■

Remark 2.3.24 (Half-open Intervals) *The index method (Def. 2.3.20) which is used in the member method returns for a given x the* largest *index i such that* $p_i.x \leq x$*. This causes that the envelope function is interpreted as a half-open interval which is closed at the left hand side and open at the right hand side.*

To see this, consider the following example:

$$I = \underbrace{(0,0)}_{p_0}\underbrace{(0,500)}_{p_1}\underbrace{(10,500)}_{p_2}\underbrace{(10,1000)}_{p_3}\underbrace{(50,1000)}_{p_4}\underbrace{(50,500)}_{p_5}\underbrace{(60,500)}_{p_6}\underbrace{(60,0)}_{p_7}$$



Half-Open Interval

We have $I.member(0) = 500$, $I.member(10) = 1000$, $I.member(50) = 500$, $I.member(60) = 0$ because $I.index(50) = 5$ and $I.index(60) = 7$. ■

Remark 2.3.25 (Extreme Cases) *There are a number of extreme cases of envelope polygons I:*

- *$I = ()$ represents the empty set;*
- *$I = ((a, 0))$ also represents the empty set (which gets normalised to $()$);*
- *$I = ((a, y))$ with $y > 0$ represents the infinite fuzzy interval with constant membership function $I(x) = y$;*
- *$I = ((a, y_1)(a, y_2))$ represents the fuzzy interval with membership function*

$$I(x) = \begin{cases} y_1 & \text{for } x < a \\ y_2 & \text{for } x \geq a. \end{cases}$$

- *$((0, 0)(0, \top))$ represents $[0, +\infty[$*
- *$((0, \top)(0, 0))$ represents $]-\infty, 0[$*

■

2.3.2.2 Basic Features of Fuzzy Intervals

We start with some simple predicates for checking whether the intervals are infinite.

Definition 2.3.26 (Infinity Predicates) *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon.*

$$\begin{array}{lll} I.isNegInfinite() & \stackrel{\text{def}}{=} & I \neq () \text{ and } p_0.y > 0 \\ I.isPosInfinite() & \stackrel{\text{def}}{=} & I \neq () \text{ and } p_n.y > 0 \\ I.isInfinite() & \stackrel{\text{def}}{=} & I \neq () \text{ and } p_0.y > 0 \text{ or } p_n.y > 0 \end{array}$$

■

Using the *indexMax*-method (Def. 2.3.20) we can define $I.sup()$ for computing the height $\sup(I)$ (Def. 2.2.3) of the fuzzy interval.

Definition 2.3.27 (sup and inf Values) *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon.*

$$I.inf() \stackrel{\text{def}}{=} min\{p_i.y \mid 0 \leq i \leq n\}$$

$$I.sup() \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } I = () \\ p_{I.indexMax(true)}.y & \text{otherwise} \end{cases}$$

The result of sup and inf are y-integer values. ■

The complexity of *sup* and *inf* are in general linear because *indexMax* requires linear search. It is constant for crisp intervals.

Size of Fuzzy Intervals

The *size* of a fuzzy interval is the integral over its membership functions (Def. 2.2.8). We define now three methods for computing the (doubled) size of a fuzzy interval. $size2()$ computes the overall size, i.e. $I.size2()/\top = 2|I|$. $I.size2(k, l)$ computes the size between two vertices of the envelope polygon, i.e. $I.size2(k, l)/\top = 2|I|_{p_k.x}^{p_l.x}$. Finally $I.size2(a, b)$ computes the size between two arbitrary x-coordinates a and b: $I.size2(a, b)/\top = 2|I|_a^b$.

Definition 2.3.28 (Size) *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon. Let k and l be two indices.*

$$I.size2I(k,l) \overset{\text{def}}{=} \begin{cases} undefined & \textit{if } k < 0 \textit{ or } l > n \\ -I.size2(l,k) & \textit{if } l < k \\ 0 & \textit{if } k = l \textit{ or } I = () \\ \Sigma_{m=k}^{l-1} p_m.area2(p_{m+1}) & \textit{otherwise} \qquad (\textit{Def.2.3.9}) \end{cases}$$

$$I.size2() \overset{\text{def}}{=} \begin{cases} 0 & \textit{if } I = () \\ +\infty & \textit{if } p_0.y > 0 \textit{ or } p_n.y > 0 \\ I.size2(0,n) & \textit{otherwise} \end{cases}$$

Both versions of $size2$ return x-integers.

Now let a and b be two x-coordinates:

$$I.size2(a,b) \overset{\text{def}}{=} \begin{cases} 0 & \textit{if } I = () \textit{ or } a = b \\ -I.size2(b,a) & \textit{if } b < a \\ 2 \cdot (b-a) \cdot p_n.y & \textit{if } a \geq p_n.x \\ 2 \cdot (b-a) \cdot p_0.y & \textit{if } b \leq p_0.x \\ (b-a) \cdot (p_i.lineY(p_{i+1}, a) + p_i.lineY(p_{i+1}, b)) & \textit{if } p_{i-1}.x \leq a \leq b \leq p_i.x \\ \qquad \textit{where } i = I.index(a) & \\ head + middle + tail & \textit{otherwise} \end{cases}$$

where

$$head \overset{\text{def}}{=} \begin{cases} 2 \cdot (p_0.x - a) \cdot p_0.y & \textit{if } a \leq p_0.x \\ 0 & \textit{if } p_i.x = a \\ p_i.area2(p_{i+1}, a, p_{i+1}.x) & \textit{otherwise} \\ \qquad \textit{where } i = I.index(a) \textit{ and} & \end{cases}$$

$$middle \overset{\text{def}}{=} I.size2(I.index(a), I.index(b)) \textit{ and}$$

$$tail \overset{\text{def}}{=} \begin{cases} 2 \cdot (b - p_n.x) \cdot p_n.y & \textit{if } b \geq p_n.x \\ 0 & \textit{if } p_i.x = b \\ p_i.area2(p_{i+1}, p_i.x, b) & \textit{otherwise} \\ \qquad \textit{where } i = I.index(b) & \end{cases}$$

The method returns a floating point value. ■

The next two methods compute the centre and middle points for a fuzzy interval (Def. 2.2.9).

Definition 2.3.29 (Centre Points) *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon.*

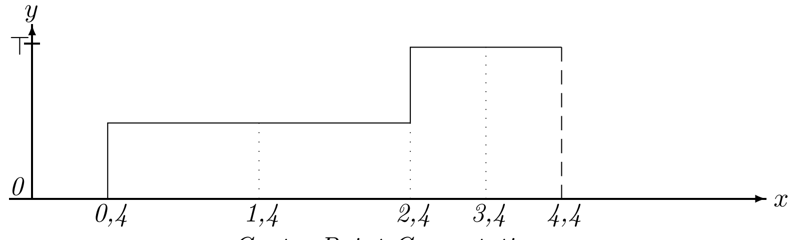
$$I.centrePoint(k,m) \overset{\text{def}}{=} \begin{cases} undefined & \textit{if } I = () \textit{ or } I.isInfinite() \\ p_0.x & \textit{if } k = 0 \\ p_n.x & \textit{if } k = m \\ p_{i-1}.area2X(p_i, \frac{s \cdot k}{m} - I.size2(0, i-1)) & \textit{otherwise} \\ \quad \textit{where } s \overset{\text{def}}{=} I.size2(0,n) \textit{ and} & \\ \quad i = \min\{i \mid m \cdot I.size2(0,i) > s2 \cdot k\} & \end{cases}$$

$centrePoint$ returns a (rounded) x-integer.

The search for the index i in $centrePoint$ causes linear complexity for both methods. ■

The *centrePoint* method needs to locate the x-coordinate such that $|I|_{-\infty}^{x} = \frac{k}{m}|I|$. To this end it first locates the index i with $p_{i-1} \leq x \leq p_i$. Then it calls the $area2X$-method to calculate the x-coordinate x with $|I|_{-\infty}^{p_{i-1}.x} + |I|_{p_{i-1}.x}^{x} = \frac{k}{m}|I|$.

Example 2.3.30 (Centre Point Computation)
Let $I = \underbrace{(0,0)}_{p_0}\underbrace{(0,500)}_{p_1}\underbrace{(4,500)}_{p_2}\underbrace{(4,1000)}_{p_3}\underbrace{(6,1000)}_{p_4}\underbrace{(6,0)}_{p_5}$



Centre Point Computation

We have $|I| = 4000$*, i.e.* $s2 = 8000$*, and we want to compute* $centrePoint(1,4)$*. The search for* $i = \min\{i \mid 4 \cdot I.size2(0,i) > 8000 \cdot 1\}$ *yields* $i = 2$ *because* $4 \cdot 4000 > 8000 \cdot 1$*.*

Since $|I|_{-\infty}^{p_1.x} = 0$ *there is still an area the size of 2000 to be covered by* $|I|_{p_1.x}^{x}$*.*
The call to $p_1.area2X(p_2, \frac{8000 \cdot 1}{4} - 0) = p_1.area2X(p_2, 2000)$ *yields 2, such that* $x = 2$ *is in fact the correct result for* $I^{1,4}$*.* ■

Components of Fuzzy Intervals
The *nComponents*-method can be used to count the number of components of an interval. It counts the number of times the envelope polygon drops down to an y-value 0 and adds 1 if it is positively infinite.

Definition 2.3.31 (Number of Components) *Let* $I = (p_0, \ldots, p_n)$ *be an envelope polygon.*

$$I.nComponents() \overset{\text{def}}{=} \begin{cases} 0 & \textit{if } I = () \textit{ or } n = 0 \textit{ and } p_0.y = 0 \\ 1 & \textit{if } n = 0 \textit{ and } p_0.y > 0 \\ \Sigma_{i=1}^{n} \begin{cases} 1 & \textit{if } p_i.y = 0 \textit{ and } p_{i-1}.y > 0 \\ 0 & \textit{otherwise} \end{cases} & + \begin{cases} 1 & \textit{if } p_n.y > 0 \\ 0 & \textit{otherwise} \end{cases} \end{cases}$$
■

The method *component*(k) below extracts from an envelope polygon the k^{th} component as a new envelope polygon.

Definition 2.3.32 (component) *Let* $I = (p_0, \ldots, p_n)$ *be an envelope polygon.*

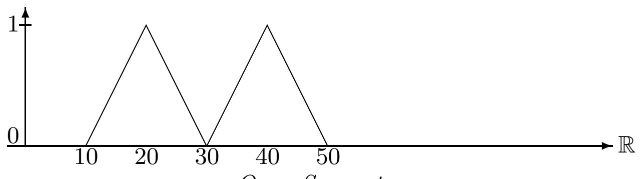
$$I.component(k) \overset{\text{def}}{=} \begin{cases} () & \textit{if } I = () \\ V_{i=I.skipComponent(k-1)}^{n} \begin{cases} (p_i) \textit{ and break} & \textit{if } p_i.y = 0 \textit{ and } p_{i-1}.y > 0 \\ (p_i) & \textit{otherwise} \end{cases} \end{cases}$$

where I.skipComponent(k) returns the first index of the $k + 1^{st}$ *component.*
It is described procedurally.

```
If $k = 0$ return 0.
If $n = 0$ return 1.
Let $l \overset{\text{def}}{=} 0$.
$For_{i=1}^{n}$ {$if(p_i.y = 0$ and $p_{i-1}.y > 0)$ $l = l + 1$;   // next component
            $if(l = k)${ $if(i = n)$ return $n + 1$          // last component skipped
                     $if(p_{i+1}.y > 0)$ return $i$       // the two components meet at $p_i.x$
                     else return $i + 1$.}}
 return $n + 1$                                     // last component skipped
```
■

2.3.2.3 Regions

A basic design decision for the algorithms in the FuTI–module is to have only half open crisp intervals of the form $[a, b[$. Unfortunately there are fuzzy intervals with open intervals as support (Def. 2.2.4). Consider the following interval:



Open Support

Its support is the open interval $]10, 30[,]30, 50[$. If this is again represented as a fuzzy interval in the FuTI style, it corresponds to the half open interval $[10, 50[$. This is a fundamental weakness of the FuTI–module, which, hopefully, has no practical relevance.

The FuTI–module has therefore the concept of a *region* of a fuzzy interval, which approximates the real regions as half-open intervals.

Definition 2.3.33 (Region) *We define an enumeration type* `Region` *with values* `core`*,* `support`*,* `kernel` *and* `maximum`*. These values indicate the smallest, possibly non-convex, half open interval which contain the corresponding regions of Def. 2.2.4.* ■

This means that if one of the subsequent functions with parameter `region` computes the corresponding region, the smallest half open interval which contain the corresponding regions of Def. 2.2.4 is meant.

The components of the regions of an interval can be enumerated with the method *nextComponent*.

Definition 2.3.34 (nextComponent) *The method* $bool I.nextComponent(x1, x2, i1, i2, region)$ *computes the components of the given region (core, support, kernel or maximum) of the interval I one by one. The first component is computed when the variable* $i1$ *has the value -1. In this case* $x1$ *and* $x2$ *are bound to the x-coordinates of the endpoints of the component.* $x1$ *and* $x2$ *may be the infinity.* $i1$ *and* $i2$ *are bound to the indices of the endpoints of the component in the envelope polygon. Successive calls to nextComponent yield the next components. The Boolean result of nextComponent is true as long as there is still a component which has just been computed.* ■

Notice that *nextComponent* computes for the above example with support $]10, 30[,]30, 50[$ indeed the two components $[10, 30[$ and $[30, 50[$, and not $[10, 50[$.

The FuTI–module provides three more methods for extracting information about the regions of an interval, *size*, *crisp* and *side*:

Definition 2.3.35 (size) *The method* $I.size(region)$ *measures the size of the given region (core, support, kernel or maximum) of the interval I in x-coordinates.* ■

The algorithm uses $I.nextComponent(x1, x2, i1, i2, region)$ (Def. 2.3.34) to enumerate the components of I's region, and adds up the differences $x2 - x1$. If one of these values is the infinity then the infinity is returned.

Definition 2.3.36 (crisp) *The method* $I.crisp(region)$ *turns the given region (core, support, kernel or maximum) into a – possibly non-convex – crisp interval.* ■

The algorithm uses $I.nextComponent(x1, x2, i1, i2, region)$ (Def. 2.3.34) to enumerate the components of I's region, and constructs from $x1$ and $x2$ crisp subintervals which are joined to a new interval.

Definition 2.3.37 (side) *The method $I.side(region, front)$ computes the left (if $front = true$) or right (if $front = false$) side of the region (core, support, kernel or maximum) of the interval I.*

The result may be the infinity. ■

2.3.2.4 Point–Interval and Interval–Interval Relations

Point–interval and interval–interval relations for fuzzy intervals should not yield boolean, but fuzzy results. Unfortunately the situation here is even worse than with the set operations for fuzzy intervals. There are no natural and obvious definitions for such relations. There are different and equally plausible versions. Therefore there are no fuzzy point–interval and interval–interval relations within the FuTI–module. Different versions of such relations have been specified in the GeTS language [34].

The FuTI–module contains instead crisp point–interval and interval–interval relations between the regions of the intervals. Since the regions are crisp intervals, there are natural definitions for these relations.

Point–Interval Relations for Regions
The relations between a single time point and a, possibly non-convex, (crisp) region of a fuzzy interval are: $before$, $starts$, $during$, $finishes$, $after$ and *between*. The point–interval relations $before$ and $after$ respect that the regions are treated as half open intervals $[a....b[$. That means $t\ before\ [a....b[$ is only true if $t < a$. $t\ after\ [a....b[$ is true if $t \geq b$.

The *between*–relation makes sense for non-convex regions of intervals. $t\ betweenI$ means that t is in a gap between the components of the corresponding region of I. Notice that for the above interval I with support $]10, 30[,]30, 50[$ we get $(30\ between\ I) = false$ because the `support` region (Def. 2.3.33) is $[10, 50[$ in this case.

Interval–Interval Relations
For crisp intervals there is the standard set of Allen's interval–interval relations (Fig. 2.1) [1].

These relations are defined for convex crisp intervals. Since the FuTI–module has to deal with non-convex crisp intervals as well, it has to generalise these relations to the non-convex case. The generalisation is straightforward for $before$, *meets*, *during* and $after$. The condition for I *overlaps* J is that a part of I is before J and the rest of I is during J, and this version is implemented in FuTI. The condition for I *starts* J is that the left ends of I and J coincide and I is during J, and this works also for non-convex intervals. The analogous conditions hold for the $finishes$ relation.

There is a further subtlety to be considered when implementing interval–interval relations. Allen's interval–interval relations come with a constraint calculus which relies heavily on the condition that the relations are disjoint. That means, two intervals I and J stand in exactly one of the relations to each other. For example, $[0, 10[$ meets $[10, 20[$, and therefore $[0, 10[$ before $[10, 20[$ must be false. The disjointness of the relations is necesary for the constraint calculus. For other applications it not very intuitive. Therefore the interval–interval relations in the FuTI–module are defined in the more intuitive way where both $[0, 10[$ meets $[10, 20[$,

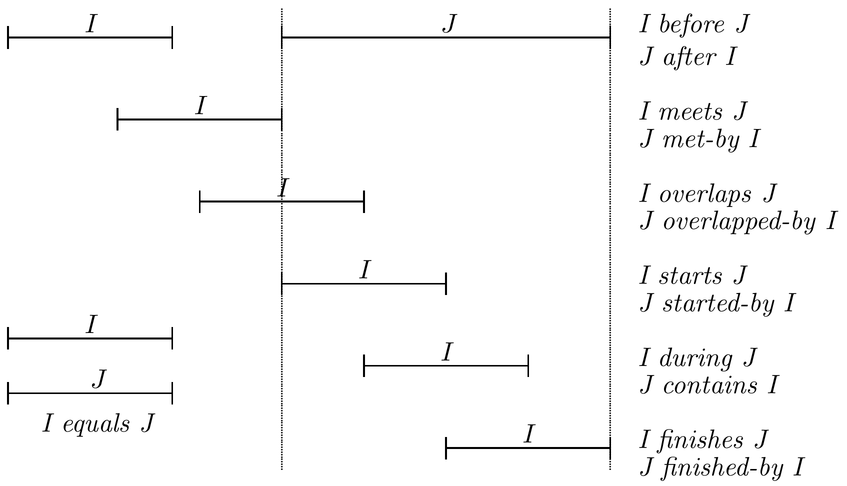


Figure 2.1: Interval–Interval Relations

and $[0,10[$ before $[10,20[$ are true. By similar reasons, the *during* relation is defined such that $starts \subseteq during$ and $finishes \subseteq during$.

The FuTI–module provides two sets of interval–interval relations. One set is between ordinary intervals $[a,b[$ and regions of fuzzy intervals. The second set is between the corresponding regions of two fuzzy intervals. The algorithms have either constant or at worst linear complexity. In particular the algorithm for *during* uses a sweep line technique for going only once through the two envelope polygons.

2.3.2.5 Hull Operators

The method $I.crispHull()$ implements the $crispHull()$-function (Def. 2.2.27).

Definition 2.3.38 (Crisp Hull) *Let $I = (p_0,\ldots,p_n)$ be an envelope polygon.*

$$I.crispHull() \overset{\text{def}}{=} \begin{cases} () & \text{if } I = () \\ \left(\begin{cases} ((p_0.x,\top)) & \text{if } p_0.y > 0 \\ ((p_0.x,0)(p_0.x,\top)) & \text{otherwise} \end{cases}, \begin{cases} ((p_n.x,\top)) & \text{if } p_n.y > 0 \\ ((p_n.x,\top)(p_n.x,0)) & \text{otherwise} \end{cases} \right) \\ \text{otherwise} \end{cases}$$

■

The method $I.monotoneHull()$ implements the $monotoneHull()$-function (Def. 2.2.29). The algorithm scans the envelope polygon first from 0 to the first maximal element and skips all vertices which destroy monotonicity. Then it scans the envelope polygon from the last element to the last maximal element and skips again all vertices which destroy monotonicity. Finally it appends the first lists with the reversed second list.

Definition 2.3.39 (Monotone Hull) *Let $I = (p_0,\ldots,p_n)$ be an envelope polygon. We describe the algorithm $I.monotoneHull()$ procedurally:*

```
If $I = ()$ return ();
Let $newI_1 \overset{\text{def}}{=} (p_0, V_{i=0}^{I.indexMax(true)} \begin{cases} ((p_{i-1}.lineX(p_i, max), max), p_i) \\ \quad \text{if } i > 0 \text{ and } p_i.y \geq max \text{ and } p_{i-1}.y < max \\ (p_i) & \text{if } p_i.y \geq max \\ () & \text{otherwise} \end{cases})$
where $max$ is the current largest y-coordinate in $newI_1$:
Let $newI_2 \overset{\text{def}}{=} (p_n, V_{i=n}^{I.indexMax(false)} \begin{cases} ((p_{i-1}.lineX(p_i, max), max), p_i) \\ \quad \text{if } i < n \text{ and } p_i.y \geq max \text{ and } p_{i-1}.y < max \\ (p_i) & \text{if } p_i.y \geq max \\ () & \text{otherwise} \end{cases})$
where $max$ is now the current largest y-coordinate in $newI_2$:
Let $newI_2 = (q_0, \ldots, q_m)$;
$For_{i=m}^{0}$ $newI_1.push\_back(q_i)$.
return $newI_1$.
```
■

The algorithm for the convex hull function *convexHull* (Def. 2.2.28) is a special version of the *Graham Scan* algorithm for arbitrary polygons. It goes from left to right through the envelope polygon and pushes all candidates for the convex hull on a stack. Wrong candidates are later popped from the stack. Since the points are already sorted, its complexity is linear.

Definition 2.3.40 (Convex Hull) *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon. We describe the algorithm $I.convexHull()$ procedurally:*

```
If $I = ()$ return ().
Let $f \overset{\text{def}}{=} \begin{cases} I.indexMax(true) & \text{if } p_0.y > 0 \\ 0 & \text{otherwise} \end{cases}$
Let $l \overset{\text{def}}{=} \begin{cases} I.indexMax(false) & \text{if } p_n.y > 0 \\ n & \text{otherwise} \end{cases}$
Let $newI \overset{\text{def}}{=} (p_{i_f})$.
$For_{i=f}^{l}$   while($m \geq 1$ and $q_{m-1}.leftturn(q_m, p_i)$) $newI.pop\_back()$;
         $newI.push\_back(p_i)$;
where $q_m$ is the current last element of $newI$.
return $newI$.
```
■

2.3.2.6 Basic Unary Transformations

A number of basic unary transformations (Def. 2.2.30) can be implemented by just manipulating the vertices of the envelope polygons.

Definition 2.3.41 (Extend, Scaleup, Shift) *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon. The three functions return $()$ if $I = ()$. Let $M \overset{\text{def}}{=} (q_0, \ldots, q_k) \overset{\text{def}}{=} I.MontoneHull()$.*

$$
\begin{aligned}
I.extend(true) &\overset{\text{def}}{=} (V_{i=0}^{j}(q_i), (q_j.x, \top)) \\
&\qquad \text{where } j = M.indexMax(true) \\[1ex]
I.extend(false) &\overset{\text{def}}{=} ((q_j.x, \top), V_{i=j}^{k}(q_i)) \\
&\qquad \text{where } j = M.indexMax(false) \\[1ex]
I.scaleUp() &\overset{\text{def}}{=} V_{i=0}^{n}(p_i.x, roundY((p_i.y \cdot \top / I.sup()))) \\[1ex]
I.shift(a) &\overset{\text{def}}{=} V_{i=0}^{n}(p_i.x + a, p_i.y)
\end{aligned}
$$

■

extend(*true*) implements $extend^+$, *extend*(*false*) implements $extend^-$, (Def. 2.2.30).

cut

We provide three *cut*-methods. The first one cuts an envelope polygon between two given x-coordinates x_1 and x_2. The second one cuts it between the x-coordinates of two given vertices. The third one cuts the interval after or before an x-coordinate.

Definition 2.3.42 (cut) *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon. x, x_1 and x_2 are x-coordinates.*

$$I.cut(x_1, x_2) \stackrel{\text{def}}{=} \begin{cases} () & \text{if } x_2 \leq x_1 \\ ((x_1, 0), (x_1, I.member(x_1)), (V_{i=I.index(x_1)}^{I.index(x_2)} p_i), (x_2, I.member(x_2)), (x_2, 0)) \\ \quad \text{otherwise} \end{cases}$$

where the list is formed with the push_back operator (Def. 2.3.18). This removes certain redundancies.

Let i_1 and i_2 be two indices.

$$I.cutI(i_1, i_2) \stackrel{\text{def}}{=} \begin{cases} () & \text{if } i_2 \leq i_1 \\ V_{i=i_1}^{i_2}(p_i) & \text{otherwise.} \end{cases}$$

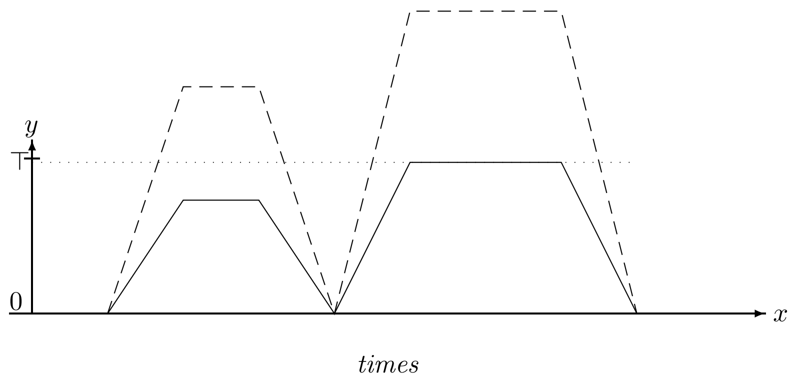
$$I.cut(x, true) \stackrel{\text{def}}{=} ((x, 0), (x, roundY(I.member(x))), V_{i=I.index(x)}^{n} p_i)$$

$$I.cut(x, false) \stackrel{\text{def}}{=} (V_0^{i=I.index(x)} p_i, (x, roundY(I.member(x))), (x, 0))$$

■

times

The *times* operator, which multiplies the membership function with a constant, is not so easy to implement. Since $y \cdot a > \top$ is possible, one has to cut the multiplied envelope polygon at $y = \top$. The picture below illustrates the problem.



times

In order to cut the multiplied polygon at $y = \top$ the intersection points between the dotted and dashed lines have to be computed. The *times* function defined below follows the line segments of the envelope polygon I and checks whether the multiplied line segments cross the $y = \top$ line. In this case the intersection points are computed and inserted into the transformed polygon.

Definition 2.3.43 (times) *Let $I = (p_0, \dots, p_n)$ be an envelope polygon and a a non-negative floating point number.*

$$I.times(a) \overset{\text{def}}{=} \begin{cases} () & \text{if } I = () \\ \left(\begin{array}{l} (p_0.x, \min(1, p_0.y \cdot a)), \\ V_{i=1}^{n} \begin{cases} () & \text{if } p_{i-1}.y \cdot a \geq \top \text{ and } p_i.y \cdot a > \top \\ ((x, \top)) & \text{if } p_i.y \cdot a < \top \text{ and } p_{i-1}.y \cdot a > \top \\ ((x, \top), (p_i.x, p_i.y \cdot a)) & \text{if } p_i.y \cdot a > \top \text{ and } p_{i-1}.y \cdot a < \top \\ & \text{where } x = p_{i-1}.lineX(p_i, \top) \\ ((p_i.x, p_i.y \cdot a)) & \text{otherwise} \end{cases} \end{array} \right) \\ \quad \text{otherwise} \end{cases}$$

■

Interpolation

Some of the transformations of fuzzy time intervals are non-linear in the sense that they transform straight lines into curved lines. These transformations cannot be implemented by simply transforming the vertices of the envelope polygons. Since the result of the transformations must be envelope polygons, we need to approximate curved lines by polygons. To this end we define a method *interpolate* which interpolates curved lines between vertices of polygons.

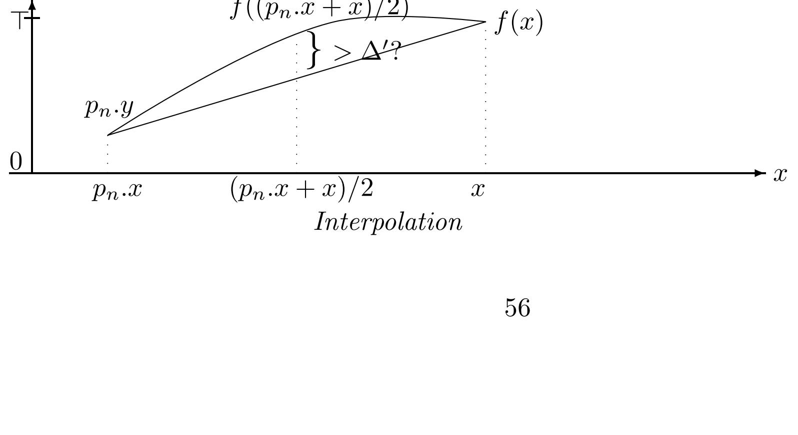
Definition 2.3.44 (Interpolation) *Let $I = (p_0, \dots, p_n)$ be a non-empty envelope polygon, x an x-coordinate, f a function from x-coordinate $\mapsto$ y-coordinate and Δ a threshold value (e.g. $\Delta = 0.1$).*

$$I.interpolate(x, f, \Delta) \overset{\text{def}}{=}$$

$$\begin{cases} I \quad \text{if } x \leq p_n.x \\ I.interpolate(roundX((p_n.x + x)/2), f, \Delta).interpolate(x, f, \Delta) \\ \quad \text{if } |2y_1 - y_2| > \Delta' y_2 \\ \quad \text{where } y_1 = f(roundX((p_n.x + x)/2)) \text{ and } y_2 = p_n.y + f(x) \text{ and } \Delta' = \Delta/(1 + 2y_2/\top) \\ I.push\_back((x, f(x))) \quad \text{otherwise} \end{cases}$$

■

The *interpolate*-method starts with an envelope polygon $I = ((x_0, y_0))$ and fills up I with interpolated values. Suppose $I = (p_0, \dots, p_n)$. For a given $x > p_n.x$ it checks whether the relative difference between the middle point $(p_n.x + x)/2$ of the straight line between p_n and $(x, f(x))$, and $f((p_n.x + x)/2)$ is larger than Δ. If this is not the case then the approximation is good enough and the point $(x, f(x))$ is pushed onto I. If this is the case, better interpolation is necessary. Therefore it calls itself recursively with x = middle point to fill up I until the middle point, and then with x itself to fill up I from the middle point until the actual x. The threshold Δ is only a basic threshold for very small y-values. The threshold Δ' causes that the interpolation becomes denser for larger y-values.



Interpolation

Integration
The *integrate*$^+$-function (Def. 2.2.30) is implemented by the *integrate*$(true)$-method below and the *integrate*$^-$-function is implemented by the *integrate*$(false)$-method. *integrate*$(true)$ goes from left to right through the envelope polygon I and calls for each line segment the *area*2-function for points (Def. 2.3.9). *integrate*$(false)$ goes from right to left through the polygon. Therefore the resulting list has to be reversed. Since line segments are linear, their integration yields a quadratic curve. Therefore interpolation is necessary.

Definition 2.3.45 (Integration) *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon and Δ the threshold. We write the function again in a procedural style.*

```
I.integrate(true):
    if $I = ()$ then return ()
    Let $newI \overset{\text{def}}{=} (p_0.x, 0)$
    $For_{i=1}^{n}$ $newI.interpolate(p_i.x, \lambda(x)(q_m.y + p_{i-1}.area2(p_i, x, true)/2), \Delta)$
            where $newI = (q_0, \ldots, q_m)$
    return $newI$.
```

```
I.integrate(false):
    if $I = ()$ then return ()
    Let $newI \overset{\text{def}}{=} (p_n.x, 0)$
    $For_{i=n}^{1}$ $newI.interpolate(p_{i-1}.x, \lambda(x)(q_m.y + p_{i-1}.area2(p_i, x, false)/2), \Delta)$
            where $newI = (q_0, \ldots, q_m)$
    return $newI$ reversed.
```
■

2.3.2.7 Y-Function Based Unary Transformations

For unary transformations of fuzzy intervals which can be generated by applying a y-function to the membership values, there is a simple algorithm scheme: if the y-function is linear, apply it to the y-coordinates of the envelope polygon; if the function is not linear, use the *interpolate* method.

Definition 2.3.46 (Unary Transformation) *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon and let f be a unary y-function and Δ a threshold value. We describe the method $I.unaryTransformation(f, \Delta)$ procedurally:*

```
    If $I = ()$ return ();
    If $f$ is linear then return $V_{i=0}^{n}(p_i.x, f(p_i.y))$.
    Otherwise:
    Let $newI \overset{\text{def}}{=} (p_0.x, f(p_0.y))$;
    $For_{i=1}^{n}$ $newI.interpolate(p_i.x, \lambda(x)f(p_{i-1}.lineY(p_i, x)), \Delta)$;
    return $newI$;
```
■

Exponentiation
The exponentiation operator *exponentiate*$_e(i)$ (Def. 2.2.30) is the first non-linear transformation we consider here.

Definition 2.3.47 (Exponentiation) *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon, e a non-negative number (the exponent) and Δ the threshold.*

$I.Exp(e) \stackrel{\text{def}}{=} I.unaryTransformation(\lambda(y)y^e, \Delta)$.

$\lambda(y)y^e$ *is not linear.* ■

Complement Operator
Another point-based transformation is the complement operator.

Definition 2.3.48 (complement) *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon, n a negation function (Def. 2.2.17) and Δ the threshold.*

$I.complement(n) \stackrel{\text{def}}{=} I.unaryTransformation(n, \Delta)$. ■

2.3.2.8 Y-Function Based Binary Transformations

Y-Function based binary transformations of fuzzy intervals are more complicated to implement because besides the vertices of the two envelope polygons their intersection points are relevant for the transformation. The intersection points my become vertices of the transformed envelope polygons. Therefore the first thing the binary transformation algorithm must do is to compute the intersection points of the two polygons. Fortunately, since the two polygons are unimonotone, this can be done with a sweep line algorithm in linear time. The result of the $IntersectionPoints$-algorithm defined below is a list $((p_0, q_0), \ldots)$ of pairs of points. The p_i are the vertices of I_1 and the intersection points between I_1 and I_2. The q_i are the vertices of I_2 and also the intersection points between I_1 and I_2. $p_i.x = q_i.x$ holds for all i.

In order to simplify the presentation of the algorithm a little bit we assume that I_1 and I_2 start at the same x-coordinates, and that both polygons have a redundant extra point p_{n+1} and q_{m+1} at the end. This saves some case distinctions at the beginning and at the end of the sweep.

Definition 2.3.49 (Intersection Points) *Let $I_1 = (p_0, \ldots, p_{n+1})$ and $I_2 = (q_0, \ldots, q_{m+1})$ be two envelope polygons such that $p_0.x = q_0.x$ and $p_n.y = p_{n+1}.y$ and $q_m.y = q_{m+1}.y$. We define the method $I_1.IntersectionPoints(I_2)$. It returns a list of pairs $((p_0, q_0), \ldots)$.*

Let $IntP \stackrel{\text{def}}{=} ()$.
Let $i \stackrel{\text{def}}{=} 0$ *and* $j \stackrel{\text{def}}{=} 0$
Let $x \stackrel{\text{def}}{=} p_0.x$ *(x is the position of the sweep line).*

```
$while(x \leq \max(p_n.x, q_m.x))\{$
    $if(i < n \text{ and } p_i.x = p_{i+1}.x)$
        $if(x = q_j.x)\{$
            $IntP.push\_back(p_i, q_j); i := i + 1;\}$
            $if(j < m \text{ and } q_j.x = q_{j+1}.x) j := j + 1;\}$
        $else\{IntP.push\_back(p_i, (x, roundY(q_j.lineY(q_{j+1}, x)))); i := i + 1;\}$
        $continue;\}$

    $if(j < m \text{ and } q_j.x = q_{j+1}.x)\{$
        $IntP.push\_back(p_i, q_j);$
        $IntP.push\_back(p_i, q_{j+1}); j := j + 1$
        $continue;\}$
    $if(x = q_j.x) IntP.push\_back(p_i, p_j)$
    $else\ IntP.push\_back(p_i, (x, roundY(q_j.lineY(q_{j+1}, x))));$

    $if(i < n)\{$
        $if(j < m)\{$
            $if(p_i.intersectsProper(p_{i+1}, q_j, q_{j+1}))\{$
                $xint := p_i.intersection(p_{i+1}, q_j, q_{j+1});$
                $IntP.push\_back((xint, roundY(p_i.lineY(p_{i+1}, xint))),$
                                 $(xint, roundY(q_j.lineY(q_{j+1}, xint)));\}$

            $if(p_i.x < q_j.x)\{x = p_{i+1}.x; i := i + 1; continue;\}$
            $if(p_i.x = q_j.x)\{x = p_{i+1}.x; j := j + 1; continue;\}$
            $x = q_{j+1}.x; continue;\}$
        $x = p_{i+1}.x; continue;\}$

    $if(j < m)\{x := q_{j+1}.x; continue;\}$
    $x := x + 1;\}$

$return\ IntP.$
```

■

We can now define the *binaryTransformation*-method. It works much like the *unaryTransformation*-method. The differences are that *binaryTransformation* first needs to compute the intersection points, and that the call to the *interpolate*-method gets as input a function which is parameterised with two line segments instead of one.

Definition 2.3.50 (Binary Transformation) *Let $I_1 = (p_0, \ldots, p_{n_1})$ and $I_2 = (q_0, \ldots, q_{n_2})$ be envelope polygons. Let f be a binary y-function and Δ a threshold value.*
We describe the method $I_1.binaryTransformation(I_2, f, \Delta)$ procedurally:

```
Let $I \overset{\text{def}}{=} ((p_0, q_0), \ldots, (p_n, q_n)) = I_1.IntersectionPoints(I_2)$
If $I = ()$ return ();
If $f$ is linear then return $V_{i=0}^{n}(p_i.x, f(p_i.y, q_i.y))$.
Otherwise:
Let $newI \overset{\text{def}}{=} (p_0.x, f(p_0.y, q_0.y))$;
$For_{i=1}^{n}\ newI.interpolate(p_i.x, \lambda(x) f(p_{i-1}.lineY((p_i, x), q_{i-1}.lineY((q_i, x)), \Delta);$
return $newI$;
```

■

2.3.3 Integration over Multiplied Intervals

The motivation for the operators in this section come from certain fuzzy relations between fuzzy intervals. There is no unique generalisation of interval–interval relations like 'before' to fuzzy intervals. One idea for the generalisation works in two steps. The first step is to define a point–interval 'before'-relation: $PIbefore(x, I)$. This can also be done in different ways. Regardless how the concrete definition is, it is always possible to define this as an operator which maps an interval to an interval: $PIbefore'(I)(x) \stackrel{\text{def}}{=} PIbefore(x, I)$. We can now generalise the point–interval 'before' relation to an interval–interval 'before'-relation $IIbefore(I, J)$ by averaging the point–interval 'before' relation over J: $IIbefore(I, J) \stackrel{\text{def}}{=} \int I(x) \cdot PIbefore'(J)(x)\ dx/N(I, J)$. $N(I, J)$ is a normalisation factor which forces the result to be a fuzzy value between 0 and 1. More details about fuzzy point–interval and interval–interval relations can be found in [34].

FuTI provides two different integration operations which can be used for these purposes. We start with an auxiliary definition, a parameterised integration over multiplied membership functions.

Definition 2.3.51 (Integration over Multiplied Intervals) *Let $I = (p_0, \ldots, p_n)$ and $J = (q_0, \ldots, q_m)$ be envelope polygons. Let a be an x-coordinate. The integral $I.integrate(J, a) \stackrel{\text{def}}{=} \int I(x - a)J(x)\ dx$ is computed as follows:*

```
If $I = ()$ or $J = ()$ then return 0;
If (I.isInfinite() and J.isInfinite()) then $undefined$;
Let $Int = 0$;
If $(p_0.x + a \leq q_0.x)$ then {$j = 0$; $i = I.index(q_0.x - a)$; $x = q_0.x$;
                                   $Int = q_0.y \cdot I.size2(p_0.x, q_0.x - a)/2$;}
else {$i = 0$; $j = J.index(p_0.x)$; $x = p_0.x + a$;
      $Int = p_0.y \cdot J.size2(q_0.x, p_0.x + a)/2$;} }
while   $(i < n$ and $j < m)${
        $Int = Int + p'_i.integrate(p'_{i+1}, q_j, q_{j+1}, x, \min(p_{i+1}.x + a, q_{j+1}.x))$;        //Def.2.3.12
        where $p'_i \stackrel{\text{def}}{=} (p_i.x + a, p_i.y)$ and $p'_{i+1} \stackrel{\text{def}}{=} (p_{i+1}.x + a, p_{i+1}.y)$
        $x = \min(p_{i+1}.x + a, q_{j+1}.x)$;
        $if(x = p_{i+1}.x + a)\ i = i + 1$;
        $if(x = q_{j+1}.x)\ j = j + 1$;}

$if(p_n.x + a \leq q_m.x) Int = Int + p_n.y \cdot J.size2(p_n.x + a, q_m.x)/2$;
$else$                    $Int = Int + q_m.y \cdot I.size2(q_m.x - a, p_n.x)/2$;
return $Int$;
```

■

Asymmetric integration integrates over the multiplied membership functions of I and J and normalises the result with the size of I.

Definition 2.3.52 (Asymmetric Integration) *Let I and J be two fuzzy polygons. I must be finite. The method*

$$I.integrateAsymmetric(J) \stackrel{\text{def}}{=} roundY(\frac{2 \cdot I.integrate(J, 0)}{I.size2()})$$

computes $\int I(x) \cdot J(x)\ dx/|I|$. ■

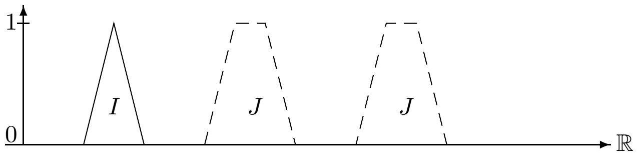
The 'symmetric integration' over multiplied envelope polygons differs from the asymmetric integration by the normalisation factor. The normalisation factor *maximizeOverlap*(I, J) below causes that there is a position of the interval I relative to the interval J such that the value of the normalised integral is 1. This is a useful operation for defining a fuzzy interval–interval 'meets' relation. It guarantees that if (a finite) I is shifted along the time axis, eventually it meets (a finite) J with resulting fuzzy value 1.

Definition 2.3.53 (Symmetric Integration) *Let I and J be two envelope polygons. The function $I.integrateSymmetric(J, simple)$ computes $\int I(x) \cdot J(x)\ dx/N$ where* $N = \begin{cases} \min(|I|, |J|) & \textit{if } simple = true \\ \max\{a \mid \int I(x-a) \cdot J(x)\ dx\} & \textit{otherwise} \end{cases}$

$$I.integrateSymmetric(J, simple)$$
$$\stackrel{\text{def}}{=} \begin{cases} undefined & \textit{if } I.isInfinite() \textit{ or } J.isInfinite() \\ roundY(\dfrac{2 \cdot I.integrate(J, 0)}{\min(I.size2(), J.size2)}) & \textit{if } simple = true \\ roundY(\dfrac{\top \cdot I.integrate(J, 0)}{maximizeOverlap(I, J)} & \textit{otherwise (Def. 2.3.56 below)} \end{cases}$$

■

The normalisation factor $maximizeOverlap(I, J) = \max\{a \mid \int I(x-a) \cdot J(x)\ dx\}$, where I is finite, amounts in general to a nontrivial search problem with unpredictable solutions. Consider the following example:



Maximising the Overlap

If we move I into the left component of J we get maximal overlap as long as I is completely contained in this part of J. The same holds for the right part of J.

For the parameter a to be maximised in the integral we get two plateaux as solutions.

There seems to be no easy analytical solution to this problem. Fortunately there are important classes of fuzzy time intervals, where this problem is extremely easy to solve.

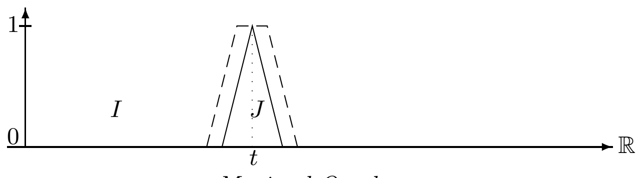
The first class is when J is infinite and $J(-\infty) = \top$ or $J(+\infty) = \top$, and, of course J has a finite kernel. In this case one can move I to the infinite part where J is constant 1. $\int I(x-a)J(x)\ dx = |I|$ in this case, i.e. $\max_a \int I(x-a) \cdot J(x)\ dx = |I|$,

The other class are the the *symmetric* and *monotone* fuzzy intervals.

Definition 2.3.54 (Symmetric and Monotone Intervals) *A fuzzy time interval I is* symmetric *if there is a time point t such that $I(t-x) = I(t+x)$ for all x holds. t is the* symmetry axis.

A fuzzy time interval I is monotone *if with increasing time coordinate x, $I(x)$ is monotonically increasing until a maximal value and then it is monotonically decreasing again.* ■

Crisp intervals are in particular monotone and symmetric. Maximal overlap is achieved for monotone and symmetric intervals if the symmetry axes of both intervals coincide.

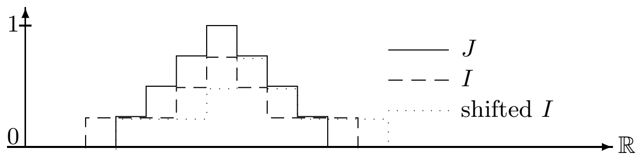


Maximal Overlap

Proposition 2.3.55 *If I and J are two monotone and symmetric fuzzy intervals then $\int I(x)J(x)\ dx$ is maximal if the symmetry axis of I and J coincide.* ■

The proof is very technical. We therefore sketch only the basic idea. First I and J are discretised into step functions with finite step size. The limit 'step size $\mapsto$ 0' is then the original problem. The discretised integral then becomes a sum $stepsize \cdot \Sigma_k I_k \cdot J_k$

One must show that moving the interval I away from the position where the two symmetry axes coincide, decreases the sum.



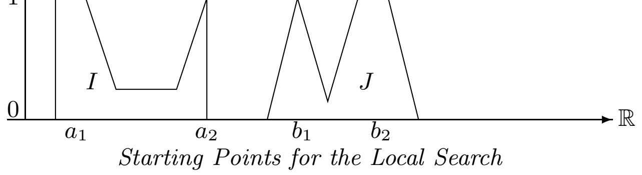
Discretised Maximisation Problem

As one can see in this picture, shifting I to the right hand side, decreases the parts of the sum $I_k \cdot J_k$ on the left side of the symmetry axis of J, and increases the parts of the sum on the right side of the symmetry axis. The important observation is, that because J is monotone falling at the right hand side, the parts I_k on the right side, which cause the sum to increase again, are multiplied with smaller J_k than the corresponding parts on the left hand side. Therefore the sum gains less on the right hand side than it looses on the left hand side. The overall sum therefore decreases or remains constant.

A General Search Procedure

We want to find a value for a such that $\int I(x-a)J(x)\ dx$ is maximal. If I or J are not monotone and symmetric a general search procedure has to be applied. The search procedure which is implemented in FuTI is a combination of an iterated binary local search with a randomised global search. It is optimised for search spaces with little structure and terminates quickly. 100% success, however, is not guaranteed.

The first problem to be solved is to find good starting points for the search. Reasonable choices are the middle points of the local maxima of I and J. For the examples in the picture below the search starts by matching the four combinations of a_k with b_l.



Starting Points for the Local Search

Since all these combinations may miss the global maximum, random start points are also generated.

The second problem is to choose an initial step size for the search. The initial step size is $\Delta = \min(J^{lS} - b_0, b_0 - J^{fS})/2$, i.e. half way between the start point b_0 of the search in the interval J and the closest end of J.

If, for example, the value for the integral increases for $a_0 + \Delta$ then the local search procedure is called recursively for the initial value $a_0 + \Delta$ and step size $\Delta/2$. The other cases are similar. This way Δ is decreased exponentially until it reaches a certain threshold. The new value for a is now the start point of another local search with the same Δ as before. This is iterated until the changes in the integral falls under another threshold (1% seemed to be a good choice).

Definition 2.3.56 (The Search for Maximising the Overlap) *Let I and J be two* finite *fuzzy intervals. We define a local search function and then a global search procedure for maximising the integral*

$$Int(a) \stackrel{\text{def}}{=} \int I(x-a)J(x)\;dx.$$

Let a be the start value for the search and Δ the step size. 'threshold' *is threshold for Δ.*

$$localSearch(a,\Delta) \stackrel{\text{def}}{=} \begin{cases} (a, Int(a)) & \text{if } \Delta \leq \text{threshold} \\ localSearch(a+\Delta, \Delta/2) & \text{if } Int(a+\Delta) > Int(a) \text{ and } Int(a+\Delta) \geq Int(a-\Delta) \\ localSearch(a-\Delta, \Delta/2) & \text{if } Int(a-\Delta) > Int(a) \text{ and } Int(a-\Delta) \geq Int(a+\Delta) \\ localSearch(a, \Delta/2) & \text{otherwise} \end{cases}$$

$iteratedLocalSearch(a,\Delta)$: iterate $(a, Int) := localSearch(a,\Delta)$ until the changes in Int falls under a threshold. return Int.

The global search procedure $maximizeOverlap(I,J)$ is described procedurally:

For all combinations m_i and n_j of middle points of local maxima of I and J:
let $\Delta = \min(J^{lS} - n_j, n_j - J^{fS})/2$, call $Int = iteratedLocalSearch(n_j - m_i, \Delta)$ and choose the maximal Int-value.

Repeat this k times with randomly chosen m_i and n_j and choose again the maximal Int-value. ($k = 5$ seemed to be enough.)

return the maximal Int-value. ■

2.4 Circular Fuzzy Intervals

A typical example for circular fuzzy intervals are angles between 0 and 360 degrees. A notion like 'around 360 degrees' could be represented by a fuzzy distribution 340,0 350,1 10,1 20,0 where 340, 350, 10, and 20 are degrees. This example shows already the difference to ordinary (linear) fuzzy intervals introduced so far: the x-coordinates need not be in increasing order.

We can reuse the ideas and algorithms for linear fuzzy intervals with a very simple trick: circular intervals are represented as ordinary intervals where the x-coordinates are normalised in a certain way such that the x-coordinates are again in increasing order. For the above example, this means that internally the interval 340,0 350,1 10,1 20,0 is stored as a polygon 340,0 350,1 370,1 380,0. This way all algorithms can exploit that the x-coordinates are in increasing order.

The circularity is taken into account by the constructor functions and the push_back method. They normalise the x-coordinates appropriately.

Despite the normalisation, the `member` method works as expected. Take again the above example: the intended interval is: 340,0 350,1 10,1 20,0, its internal representation is 340,0 350,1 370,1 380,0. Nevertheless, `member(10)` and `member(370)` both yield 1.

2.5 Summary

This report is a detailed description of the FuTI–module. This module is a C++-package for representing and manipulating fuzzy time intervals. The mathematical background, the concrete data structures and algorithms, and the interface to the module is described. The FuTI–module is used in the GeTS language [32]. This language in turn is then used to define point–interval and interval–interval relations for fuzzy intervals [34].

Appendix A contains the detailed interface description of the FuTI–module.

Chapter 3

Periodic Temporal Notions – The PartLib–Module

3.1 Introduction

The basic time units of calendar systems, years, months, weeks, days etc. are the prototypes of periodic temporal notions. Because time is one of the most important parameters of our life, the representation of temporal notions, and in particular periodic temporal notions, is necessary in many computer applications. There have been quite intensive studies of periodic temporal notions from various points of view. One can distinguish at least three approaches.

First of all, there is the important work of Dershowitz and Reingold [13] who analysed existing calendar systems and came up with algorithms for converting date information from one system to another. These algorithms are the basis for the implementation of concrete calendar systems in computer programs.

On a more abstract level there is all the work about the mathematical representation of periodic temporal notions as *time granularities*, or similar kind of mathematical objects. A good overview is given in the book of Bettini, Jajoda and Wang [5]. This work is particularly motivated by the need to represent time in temporal databases. A selection of papers about the abundant work in this area is [3, 27, 21, 38, 22, 26, 15, 4, 16, 6, 20, 7]. Since time granularities are the most important objects in this area, we introduce them already at this early place in the paper. A time granularity is usually defined as a mapping of a subset of the integers to sets of intervals in the time domain, the *granules*. This mapping must have certain properties in order to count as time granularity. Another way to explain time granularities is: a *granule* is a, possibly non-convex finite subinterval of the time domain. A *time granularity* is a sequence of such granules. One can require that this sequence is consecutive, i.e. the rightmost time point of a granule n comes before the leftmost time point of the granule $n + 1$. Sometimes, however, overlapping granules are also considered [16]. The simplest time granularities are in fact partitionings of the time domain. All basic time units, years, months etc., are of this type. The granules consist of one single interval, and there are no gaps between them. Granules consisting of one single interval only, but with gaps between them, can, for example, be used to model 'weekend'. The time spans between the weekends are the gaps between the granules. Granules consisting of several intervals are useful to model notions like 'my working day', where there is a lunch break which should not count as part of 'my working day'. Overlapping granules might be used to model, for example, the union of 'my working day' and 'my wife's working day'. The 'time granularity community' has developed ways for constructing time granularities, usually as algebraic operations on previously constructed time granularities. Conversion operations between different granularities have been defined. Relations between different time granularities have been developed, and applications, mainly in the area of temporal databases, have been considered.

An even further abstraction is possible by axiomatising temporal notions in an expressive enough logic, for example in first order predicate logic. The SOL time theory (SOL for Structured Temporal Object) of Diana Cuckierman with a first order formalisation of time loops is a prominent example for this approach [10, 11, 12].

This document presents an alternative to the granularities approach found in the literature. Periodic temporal notions are presented at a basic level as partitionings of the real numbers, which is the simplest form of granularities. To compensate for this very weak structure, names (labels) are introduced for the partitions. The labels carry information about the meaning of the partitions. Labelled partitionings can be structured in quite different ways. Therefore PartLib introduces the concept of *granularities* as a means to structure labelled partitionings.

3.1.1 Peculiarities of Real Time Systems

PartLib supports the following phenomena of real time measurement

1. Leap seconds: in order to compensate for the slowing down of earth' rotation almost every other year since 1971 the last minute of the year is made up to 10 seconds longer.

2. Daylight Savings Time Schemes: the day, week and month partitioning of a calendar system can be modified by incorporating the Daylight Savings Time rules of a given region. To this end the notion of a DLS-region is introduced, which contains the daylight savings time rules of a particular region. One such DLS-region is West Germany, where daylight savings time was introduced already in 1916.

3. Complex application specific partitionings: Examples for these are the Easter time, which depends on the moon phases, tides, sunrises and sunsets, celestic phenomena etc. All these can be incorporated by providing suitable algorithms which map partition intervals to partition coordinates and back.

3.2 Basic Concepts of PartLib: An Introduction

This section gives a very brief introduction to the basic ideas and concepts realised in PartLib. We distinguish six levels:

1. The basic level is the physical reality of time. This involves relativistic effects, quantum effects at very small time scales, cosmologic phenomena like the big bang or big crunch.etc.

2. Relativistic, quantum and cosmological effects do not play a role for every day temporal notions. For the purposes of PartLib, it is therefore sufficient to take the time line as *isomorphic to the real numbers.* Since the most precise clocks developed so far, atomic clocks, measure the time in discrete units, it is even sufficient to restrict the representation of concrete time points to *integers*[1].

3. Most mathematical objects which are used to represent periodic temporal notions are infinite. Since infinite objects cannot be represented in a computer, we need to develop a finite representation of these infinite structures. The finite representations can still be purely mathematical concepts, but they are very close to concrete data structures in computers.

4. At this level there are the internal data structures for the temporal notions in PartLib. They are not described in this document, but in the Doxygen generated implementation documentation.

5. The fifth level, finally, is the persistence and interface level. We need a description of the temporal notions (usually in XML) which can be stored on a file and which can be turned into the internal data structures.

[1]In the standard setting these integers count seconds, and they are internally represented by 64-bit long integers. If more precise time information is to be processed, and calendar systems with sub-seconds (milliseconds, microseconds etc.) are to be used, the system automatically switches to a resolution which is fine enough to represent these time units. For these cases a version of PartLib is available where the basic time units are represented by multiple precision integers.

6. There is a further level, the (graphical) user interface level, which has not yet been developed.

3.2.1 Structures along the Time Axis

Periodic temporal notions can have a very simple structure, like, for example, the sequence of seconds measured by an atomic clock. Most periodic temporal units, however, are much more complicated and may depend on various influences. Easter time, for example, depends on the moon cycle. My working hours, as another example, depends on the week and the day structure. They may have gaps during lunch break and they may be interrupted by private and public holidays. In order to model all these phenomena, PartLib uses three levels of structures: *Partitionings* of the time axis, *Labels* and *granularities.*

3.2.1.1 Partitionings of the Time Axis

Many basic time units, seconds, minutes, hours, weeks, months, years etc. just partition the time axis into non-overlapping adjacent sequences of intervals. From a mathematical point of view these are just partitionings of the real numbers $\mathbb{R}$. Therefore the basic concept for representing periodic temporal notions in PartLib are partitionings of the real numbers into *finite* partitions. PartLib can model infinite partitionings like the infinite sequence of days, but also finite partitionings like, say, the sequence of bus stops at a particular bus station during the year 2007. That means, a partitioning in PartLib is a sequence of finite adjacent non-overlapping intervals which is either infinite, or which can be preceded or followed, or both, by an infinite partition $]-\infty, t]$ or $]t, +\infty[$.

Definition 3.2.1 (Partitions and Partitionings) *A* partitioning *of the time line in PartLib is a sequence*

$$\ldots[t_{-1}, t_0[, [t_0, t_1[, [t_1, t_2[, \ldots$$

of half open intervals in $\mathbb{R}$*, the* partitions*.*

The sequence is infinite at one or both ends, or it is preceded by an infinite interval $]-\infty, t[$ *(the* start partition*) or/and it is ended by an infinite interval* $[t, +\infty[$ *(the* end partition*).*

The boundaries t_i *are integer values* $t_i \in$ `Time`*.* ■

Since the partitions in a partitioning do not overlap, one cannot use closed intervals because the endpoints of the closed intervals would be in two different partitions. Open intervals can not be used either because then the infima and suprema of the intervals would not be in any partition at all. Therefore only half open intervals can be used, either of the type $[a, b[$, or of the type $]a, b]$. In most cases there is no preference for either of the two types, but both types should not be used together. In this document we therefore use the first type $[a, b[$.

3.2.1.2 Labels for Partitions

Many every-day periodic temporal notions correspond to partitionings whose partitions have names. The month partitioning, for example, has partitions with names “January”, “February” etc. The day partitioning has partitions with names “Monday”, “Tuesday” etc. These names are in general very useful for addressing particular partitions. Therefore PartLib has the possibility to attach *labels* (names) to partitions. This can be done in different ways. The simplest way is to specify a finite list of labels, e.g (Monday, Tuesday, ..., Sunday) and to associate the first label in this list with a particular partition. The labelling is then automatically extrapolated to the past and the future. There are other ways for attaching labels to partitions, in particular when new partitionings are defined on top of existing component partitionings. This is the case, for example, for Intersection Partitionings (Sec. 3.3.8.4) and Tree Partitionings (Sec. 3.3.8.6).

It is also possible to leave particular partitions without a label at all. For example, a labelling for "working day" based on the hour partitioning could attach the label "working hour" at all hours between 8 AM and 12 AM and between 1 PM and 6 PM. The remaining hours do not get a label. As another example, assume a partitioning models a bus timetable for a particular bus station. Different buses stop there for 2 minutes every hour between, say, minute 10 and minute 12. These two-minute intervals could be labelled with the bus number. The intervals in between do not get a label. The bus number, in turn, could be used as an index in a database of buses.

Label Hierarchies. Many concepts in everyday life can be part of a taxonomy, and the taxonomy is hierarchically structured. For example, one can subsume the days Monday - Friday under "working days" and Friday and Saturday under "weekend days". This is also possible with PartLib's labels. They can be part of a tree like hierarchy of labels. Labelled partitions can be accessed from all levels of the label hierarchy.

Definition 3.2.2 (Labels and Label Hierarchies)

- *A* Label *in PartLib is any object with a* name *(a string).*
  $\emptyset$ stands for the empty label (no label).

- *A* Label Hierarchy *LH in PartLib is a finite forest whose nodes are labels and whose edges represent the sub-label hierarchy. A label hierarchy does not contain two or more occurrences of the same label. A Label Hierarchy can also consist of just a set of isolated labels.*

  We write $l_0 \leq_{LH} l_1$ if either $l_0 = l_1$ or the label l_0 is below l_1 in the label hierarchy LH.

Let `LABEL` *be the set of all labels.* ■

3.2.1.3 granularities

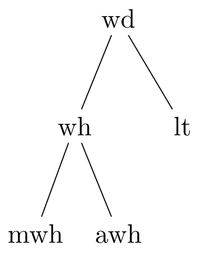
Many periodic temporal notions are not consecutive sequences of time intervals. They are sequences of intervals, but there may be gaps between them and they may even be non-convex. An example could be "my working day", which lasts from 8 AM until 6 PM with a one hour lunch break at noon. A particular working day is usually taken as a single unit, in this case an interval with an internal gap. This kind of time interval sequences is called in the literature *granularity.* A single interval in such a sequence is a *Granule.*

Definition 3.2.3 (Granules and Granularities) *A granularity is a sequence of non-overlapping, possibly non-convex, time intervals, the* granules.

A granularity G is partitioning based *iff there is a partitioning P_G such that all granules in G consist of partitions in P_G.* ■

Most approaches for modelling periodic temporal notions found in the literature use granularities as the basic concept. In contrast to this, PartLib takes the mathematically simpler partitionings as the basic concept and defines *partitioning based* granularities on top of them.

Let us take again the "my working day" example to illustrate this construction. Suppose the hours between 8 AM and 12 AM are labelled "mwh" (morning working hour), the hour between 12 AM and 1 PM is labelled "lt" (lunch time) and the hours between 1 PM and 6 PM are labelled with "awh" (afternoon working hour). The labels can be part of a label hierarchy

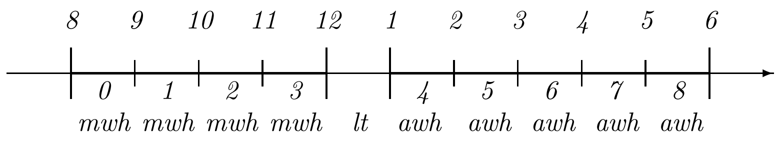


Now it is possible to define various granularities:

1. "my working day", which comprises all partitions labelled in the "wd" part of the label hierarchy (this includes lunch time);
2. "my working hours", which comprises all partitions labelled in the "wh" part of the label hierarchy (this excludes lunch time);
3. "my lunch time", which comprises all partitions with label "lt";
4. "my free time", which comprises all partitions without a label.

The working day example is used quite frequently in this document.

Example 3.2.4 (Working Day Granularity) *A "working day" lasts between 8 AM and 6 PM with a one hour lunch break at noon.*



The hours between 8 AM and 12 AM are labelled "mwh" (morning working hour), the hour between 12 AM and 1 PM is labelled "lt" (lunch time) and the hours between 1 PM and 6 PM are labelled with "awh" (afternoon working hour).

The numbers 0 - 8 count the working hours. ■

3.3 Compact Mathematical Models

3.3.1 Partition Coordinates and Integer Structures

Partitionings and granularities are usually infinite structures, and infinite structures cannot be represented in a computer with finite memory. Therefore it is necessary to find a finite representation for the infinite partitionings. An important observation here is the fact that partitionings of $\mathbb{R}$ with finite partitions can be *enumerated.* They are isomorphic to the integers.

Astronomers, for example, have long exploited this by identifying days with their *Julian Day Numbers* (JDN). This is the integer number of days that have elapsed since the initial epoch defined as noon Universal Time, Monday, January 1, 4713 BC in the proleptic Julian calendar. That noon-to-noon day is counted as Julian day 0.

A finite representation of the (infinite) concept of days can now be given by a first procedure that takes a JDN n and computes the exact start and end time of the partition that corresponds to that day, and a second procedure which takes a time point t and returns the JDN of the day containing t.

Unfortunately it turns out that there are cases where integer coordinates are just too cumbersome for identifying particular partitions of a given partitioning. For identifying days, for example, humans do not use Julian Day Numbers. Instead they count days relative to months, and months relative to years.

The mapping between partition coordinates and partitions may become much simpler and more efficient if it does not work with integer coordinates but with other more suitable data structures. This need not be, but it can be the case when new partitionings are defined on top of existing partitionings. Examples are the set theoretic composition of partitionings, for example, the intersection of the Roman catholic public holidays with the protestant public holidays. In this case one must define coordinates for the combined partitioning on top of the coordinates of the component partitionings. Since one cannot use the same integer coordinates for all three partitionings, a more appropriate data structure is necessary for the new partitioning which combines the information contained in the two component partitionings.

To overcome these problems, PartLib introduces *Integer Structures* as a representation for the general concept of isomorphisms with the integers. Different instances of this general concept can then serve as partition coordinates for particular partitionings.

Definition 3.3.1 (Integer Structure) *An Integer Structure $IS = (Co, zero, inc, dec)$ consists of*

- *a set Co of objects which are isomorphic to the integers, or at least to a connected subset of the integers,*
- *a distinguished element $zero \in Co$ which corresponds to $0 \in \mathbb{N}$, and*
- *partial or total injective increment and decrement functions inc and dec of type $Co \mapsto Co$.*

■

An integer structure need only be isomorphic to a connected subset of the integers, for example $[i, j]$ or $[i, +\infty[$ or $]-\infty, i[$ for some numbers i and j. The increment and decrement functions are undefined for the boundary values. This reflects the decision to allow partitionings with infinite start and end partitions (Def. 3.2.1).

Definition 3.3.2 (Convenient Notations for Integer Structures)
Let $IS = (Co, zero, inc, dec)$ be an integer structure. A convenient notation is:

$$IS.Co \overset{\text{def}}{=} Co \qquad IS.inc \overset{\text{def}}{=} inc$$
$$IS.zero \overset{\text{def}}{=} zero \qquad IS.dec \overset{\text{def}}{=} dec$$

Since integer structures are isomorphic to the integers or parts of them, we can use the usual operations on integers also on the integer structures.

These are in particular: $+_{IS}$, $-_{IS}$, $<_{IS}$, $>_{IS}$, $\leq_{IS}$, $\geq_{IS}$, min_{IS}, max_{IS},

The operations may take values from integer structures as well as integers directly. The main difference to the ordinary integers is that the operations on integer structures may be partial.

For a value $i \in Co$ *let* $\mathbb{I}(i)$ *be the corresponding integer under the isomorphism.* ■

The ordinary integers are of course the prototypes of integer structures.

Definition 3.3.3 (Integers as Integer Structures) *The structure* $IS_{\mathbb{I}} \overset{\text{def}}{=} (\mathbb{I}, 0, +1, -1)$ *is the integer structure (Def. 3.3.1) which consists of the ordinary integer values.* ■

There are two partitioning types in PartLib, Intersection Partitionings (Sec. 3.3.8.4) and Tree Partitionings (Sec. 3.3.8.6), where the integers themselves are not appropriate as coordinates. The easiest way to identify these partitions is by just listing their boundary values. The increment and decrement functions then need to compute the boundaries of the next/previous partition. If the partitioning is labelled then the partition coordinates need to contain besides the boundary values also a label.

Definition 3.3.4 (Partitions as Integer Structures) *A* Partition Integer Structure

$$IS_P \overset{\text{def}}{=} (Bounds, zero, inc, dec)$$

has coordinates $c \in Bounds$ *with* $c = (P, lower, upper, l)$ *such that*

1. P *is a Coordinate Based Partitioning Representation (Def.3.3.6)*
2. $[lower, upper[$ *is a partition of the partitioning* P *and*
3. l *is either a label or* $l = \emptyset$.

zero is a particular partition.

The inc and dec functions are generated by a variant of the partition coordinate *function PaCo for the partitioning P (Def. 3.3.11) as follows*

1. $inc((P, lower, upper, l)) \overset{\text{def}}{=} (P, upper, upper', l')$
   where $(P, lower', upper', l') = P.PaCo(upper)$ *and* $upper \neq +\infty$

2. $dec((P, lower, upper, l)) \overset{\text{def}}{=} (P, lower', lower, l')$
   where $(P, lower', upper', l') = P.PaCo(lower - 1)$ *and* $lower \neq -\infty$

3. $Bounds \overset{\text{def}}{=} \{zero\} \cup \{inc^n(zero) \mid n > 0 \wedge inc^n(zero) \text{ is defined}\}$
   $\cup \ \{dec^n(zero) \mid n > 0 \wedge dec^n(zero) \text{ is defined}\}$.

■

A partitioning that uses Partition Integer Structures (Def. 3.3.4) gets some of the access functions for partitions for free.

Definition 3.3.5 (Access Functions for Partitions with Partition Integer Structures) *A partitioning P which uses Partition Integer Structures (Def. 3.3.4) as partition coordinates has the following access functions for a coordinate* $i \overset{\text{def}}{=} (P, lower, upper, l)$:

$$\begin{array}{lcl} P.sPa(i) & \overset{\text{def}}{=} & lower \\ P.ePa(i) & \overset{\text{def}}{=} & upper \\ P.Label(i) & \overset{\text{def}}{=} & l \end{array}$$

■

3.3.2 Compact Representation of Partitionings

Infinite partitionings of the time line are no suitable bases for concrete algorithms operating on concrete partitionings. Therefore a compact finite representation is introduced. It is still abstract enough to cover many different partitioning types, and it has a concrete counterpart in the implementation.

The compact representation consists of an integer structure (Def. 3.3.3) for the partition coordinates and a function sPa (start of partition) which computes for a given partition coordinate the left boundary of the partition with this coordinate. Concrete partitioning types (Sec. 3.3.7) can then be realised by specifying concrete integer structures as coordinates and a concrete sPa function.

The integer structure and the sPa function are a sufficient basis for all kinds of other functions for partitionings. Some of them are listed in Def. 3.3.11 below. The definitions given there are default definitions. Their concrete implementation might become much more efficient for concrete partitioning types if they take into account the concrete information for the concrete partitioning type.

Definition 3.3.6 (Coordinate Based Partitioning Representations (CBPR)) *A Coordinate Based Portioning Representation $P = (CO, sPa)$ consists of*

- *an integer structure CO (Def. 3.3.1), the* partition coordinates*,*
- *a "start of partition" function $sPa : CO.Co \mapsto \texttt{Time}_\infty$ which maps partition coordinates to the start time of the partitions,*

such that: $sPa(i) < sPa(i +_{Co} 1)$ holds for all coordinates i where $CO.inc(i)$ is defined.

■

Definition 3.3.7 (Convenient Notations for CBPR) *If $P = (CO, sPa)$ is a Coordinate Based Portioning Representation then let*

$P.CO \overset{\text{def}}{=} CO$ *and* $P.sPa \overset{\text{def}}{=} sPa$

■

Definition 3.3.8 (Generated Partitioning) *A Coordinate Based Portioning Representation $P = (CO, sPa)$ generates the following partitioning of the time line:*

$$\begin{array}{lll} Partitions & \overset{\text{def}}{=} & \{[sPa(zero +_{CO} n), sPa(zero +_{CO} (n+1))[\mid n \geq 0, +_{CO} \textit{ is defined}\} \cup \\ & & \{[sPa(zero -_{CO} (n-2)), sPa(zero -_{CO} (n-1))[\mid n \geq 0, -_{CO} \textit{ is defined} \end{array}$$

■

Proposition 3.3.9 (The Partitioning Generator is Well Defined)
The intervals in Def. 3.3.8 for the CBPR $P = (CO, sPa)$ yields a partitioning of the time line.

Proof:

1. The intervals $[sPa(zero +_{CO} n), sPa(zero +_{CO} (n+1))[$ and $[sPa(zero -_{CO} (n-2)), sPa(zero -_{CO} (n-1))[$ are non-empty semi-open intervals because of the condition $sPa(i) < sPa(CO.inc(i))$ in Def. 3.3.6.
2. Subsequent intervals do not overlap because $sPa(zero +_{CO} (n+1))$ and $sPa(zero -_{CO} (n-1))$ respectively, is the end of one interval and the start of the next interval.

3. By the same reason, subsequent intervals do not have gaps between them.

■

Remark 3.3.10 (CBPR = Partitioning) *In the sequel we shall not distinguish any more between Coordinate Based Portioning Representations and their generated Partitionings.* ■

Definition 3.3.11 (Derived Functions for Partitionings) *Let $P = (CO, sPa)$ be a Coordinate Based Partitioning Representation.*
For a time point t and a partition coordinate i let

- $P.PaCo : \texttt{Time} \mapsto CO.Co :$ *with*

$$P.PaCo(t) \overset{\text{def}}{=} max_{CO}\{i \mid sPa(i) \leq t\}$$

  be the partition coordinate *function which maps a time point t to the coordinate of the partition containing t. (This definition is not directly implementable. Definitions which yield efficient implementations are given for concrete partitioning types.)*

- $P.ePa : CO.Co \mapsto \texttt{Time}_\infty$ *with*

$$P.ePa(i) \overset{\text{def}}{=} \begin{cases} P.sPa(i +_{CO} 1) & \text{if } i +_{CO} 1 \text{ is defined} \\ +\infty & \text{otherwise} \end{cases}$$

  be the end of the partition with coordinate i;

- $P.sPa : \texttt{Time} \mapsto \texttt{Time}_\infty$ *with*

$$P.sPa(t) \overset{\text{def}}{=} P.sPa(P.PaCo(t))$$

  be the start of the partition containing the time point t;

- $P.ePa : \texttt{Time} \mapsto \texttt{Time}_\infty$ *with*

$$P.ePa(t) \overset{\text{def}}{=} ePa(P.PaCo(t))$$

  be the end of the partition containing the time point

- $P.lengthT : \texttt{Time} \mapsto \texttt{Time}_\infty$ *with*

$$P.lengthT(t) \overset{\text{def}}{=} P.ePa(t) - P.sPa(t)$$

  be the length of the partition containing the time point t;

- $P.lengthT : CO.Co \mapsto \texttt{Time}_\infty$ *with*

$$P.lengthT(i) \overset{\text{def}}{=} P.ePa(i) - P.sPa(i)$$

  be the length of the partition with coordinate i. ■

Notice that the functions are overloaded (see Sec. 1.6). We use the same function names, regardless whether the arguments are time points or partition coordinates. The type of the argument determines the meaning of the function. The PartLib implementation uses different types for time points and partition coordinates. Therefore this is not a problem

The length functions may in fact return the infinity if applied to infinite start or end partitions. PartLib has provisions for dealing with this special case. The next example illustrates the functions.

Example 3.3.12 (for the auxiliary functions)
For the following partitioning P

$$\begin{array}{cccccc}]-\infty,-100[& [-100,0[& [0,100[& [100,101[& [101,500[& [500,+\infty[\\ -2 & -1 & 0 & 1 & 2 & 3 \end{array}$$

we have:

$$\begin{array}{lclllcl} P.PaCo(-200) & = & -2 & & P.PaCo(200) & = & 2 \\ P.sPa(-200) & = & -\infty & & P.sPa(200) & = & 101 \\ P.ePa(200) & = & 500 & & P.sPa(2) & = & 101 \\ P.ePa(2) & = & 500 & & P.lengthT(200) & = & 399 \\ P.lengthT(2) & = & 399 & & P.lengthT(-200) & = & \infty \\ P.lengthT(-2) & = & \infty & & & & \end{array}$$

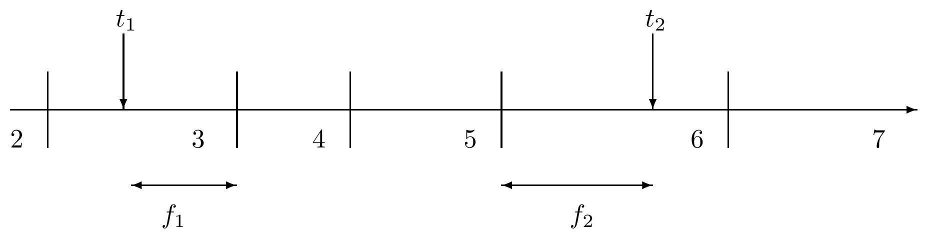
■

3.3.2.1 Length of Intervals and Time Shifts in Terms of Partitions

It is very common to measure the length of intervals or the distance between time points in terms of time units. Examples are 'The train A arrives in the station 5 minutes before the train B leaves it'. 'Tomorrow I go on a adventure trip and will be back in 3 months time'.

A very useful function is therefore $P.lengthP(t_1, t_2)$, which measures the length of the distance between t_1 and t_2 in terms of partitions of the partitioning P. $month.lengthP(t_1, t_2)$, for example, measures the length of $[t_1, t_2[$ in months. Since partitions may have different lengths, this is a nontrivial operation.

The idea for the method can be illustrated with the following picture



The distance between t_1 and t_2 is the sum of the relative length of f_1, measured as a fraction of the length of partition 3, plus the relative length of f_2, measured as a fraction of the length of partition 6, plus the number of partitions in between.

Definition 3.3.13 (Length in Partitions) *Let $P = (CO, sPa)$ be a partitioning.*
For two time points t_0 and t_1 we define a function lengthP : $\texttt{Time} \times \texttt{Time} \mapsto \mathbb{R}$.
If $t_0 \leq t_1$ then

$$P.lengthP(t_0,t_1) \overset{\text{def}}{=} \begin{cases} \dfrac{t_1 - t_0}{P.lengthT(t_0)} & \text{if } P.PaCo(t_0) = P.PaCo(t_1) \\ \mathbb{I}(P.PaCo(t_1) -_{Co} P.PaCo(t_0)) - 1 + \\ \dfrac{P.ePa(t_0) - t_0}{P.lengthT(t_0)} + \dfrac{t_1 - P.sPa(t_1)}{P.lengthT(t_1)} & \text{otherwise} \end{cases}$$

If $t_1 < t_0$ then $P.lengthP(t_0,t_1) \overset{\text{def}}{=} -P.lengthP(t_1,t_0)$.

■

Notice that $P.lengthP(t_0,t_1) = 0$ if t_0 and t_1 lie in an infinite partition.

$P.lengthP(t_1,t_2)$ is continuous. That means if t_1 is kept fixed and t_2 is moved, or the other way round, then $P.lengthP(t_1,t_2)$ makes no jumps. It is, however, not differentiable at the points where t_2 crosses the boundaries of neighbouring partitions with different length.

$P.lengthP(t_1,t_2)$ can be used to measure the absolute length of the interval $[t_1,t_2[$ if P is the partitioning for seconds or smaller time units. If P is the partitioning for minutes we can get the effect that an interval of 60 seconds length is smaller than one minute. This is the case for those minutes which contain leap seconds. Similar things happen for the coarser time units. We may get $day.lengthP(t_1,t_2) < 1$ even if $hour.lengthP(t_1,t_2) = 24$. This happens when daylight savings time is disabled just during the interval $[t_1,t_2[$, and the day is 25 hours long.

Partition Oriented Shift Function: Closely related to the length function is the *partition oriented shift function $P.shiftP(t,m)$* for partitionings. The shift function moves a time point t by m partitions forward (or backwards if $m < 0$) such that $P.lengthP(t, P.shiftP(t,m)) = m$.

Definition 3.3.14 (Partition Oriented Shift Function) *Let $P = (CO, sPa)$ be a partitioning. We define a* partition oriented *time shift function $P.shiftP(t,m)$: `Time` $\times \mathbb{R} \mapsto$ `Time`:*
Case $m \geq 0$*:*
Let $k \overset{\text{def}}{=} \dfrac{P.ePa(t) - t}{P.lengthT(t)} \in \mathbb{R}$. k is the relative distance between t and the end of the partition containing t.

$$P.shiftP(t,m) \overset{\text{def}}{=} \begin{cases} t + \lfloor m \cdot P.lengthT(t) \rfloor & \text{if } m \leq k \quad \text{(Def. 3.3.11)} \\ sPa(i) + \lfloor (m' - \lfloor m' \rfloor) \cdot P.lengthT(i) \rfloor & \text{otherwise} \\ \quad \text{where } m' \overset{\text{def}}{=} m - k & \\ \quad \text{and } \ i \overset{\text{def}}{=} P.PaCo(t) +_{CO} (\lfloor m' \rfloor + 1) & \end{cases}$$

Case $m < 0$*:*
Let $k \overset{\text{def}}{=} \dfrac{P.sPa(t) - t}{P.lengthT(t)} \in \mathbb{R}$ ($k < 0$).

$$P.shiftP(t,m) \overset{\text{def}}{=} \begin{cases} t + \lfloor m \cdot P.lengthT(t) \rfloor & \text{if } k \leq m \\ P.ePa(i) + \lfloor (m' - \lfloor m' \rfloor) \cdot P.lengthT(i) \rfloor & \text{otherwise} \\ \quad \text{where } m' \overset{\text{def}}{=} m - k & \\ \quad \text{and } \ i \overset{\text{def}}{=} P.PaCo(t) -_{CO} (\lfloor m' \rfloor + 1) & \end{cases}$$

Notice that the rounding is necessary because m can be an arbitrary real number, but the time points in `Time` are integers. ■

The shift function works for fractional shift parameters as well, but the resulting time point may get rounded to the next integer value of the integer data type which is used for the reference time. Therefore the equation $\text{length}_P(t, \text{shift}_P(t,m)) = m$ holds in fact only approximatively.

Proposition 3.3.15 *For the length function (Def. 3.3.13) and the partition oriented shift function (Def. 3.3.14) we have, up to rounding errors: $P.lengthP(t, P.shiftP(t, m)) = m$ for every time point t and every real number m.*

Proof: Case $m \geq 0$:
Subcase $m \leq k$:

$\Rightarrow \quad m \leq \frac{P.ePa(t)-t)}{P.lengthT(t)}$ Def. of k
$\Rightarrow \quad m \cdot P.lengthT(t) \leq P.ePa(t) - t$
$\Rightarrow \quad t + m \cdot P.lengthT(t) \leq P.ePa(t)$
$\Rightarrow \quad PaCo(t) = PaCo(P.shiftP(t))$ Def. of $shiftP$
$\Rightarrow \quad P.lengthP(t, P.shiftP(t)) = P.lengthP(t, t + m \cdot P.lengthT(t)) = \frac{t+m\cdot P.lengthT(t)-t}{P.lengthT(t)} = m$

Subcase $m > k$: Let $t' \overset{\text{def}}{=} P.shift(t, m)$
$\Rightarrow i > 1$
$\Rightarrow P.PaCo(t) < P.PaCo(t')$
$\Rightarrow P.PaCo(i) = PaCo(P.sPa(i) + \lfloor m' - m \rfloor \cdot P.lengthT(i))$ since $\lfloor m' - m \rfloor < 1$
$\Rightarrow P.sPa(i) = P.sPa(t')$

$$\Rightarrow P.lengthP(t,t') = \mathbb{I}(\underbrace{\underbrace{P.PaCo(t')}_{i} -_{CO} \underbrace{P.PaCo(t))}_{i-\lfloor m-k \rfloor -1} -1}_{\lfloor m-k \rfloor} +k + \underbrace{\frac{t' - P.sPa(t')}{P.lengthP(t')}}_{m-k-\lfloor m-k \rfloor} = m$$

because $\frac{t'-P.sPa(t')}{P.lengthP(t')} = \frac{P.sPa(i)+(m'-\lfloor m' \rfloor)\cdot P.lengthT(i)-P.sPa(t')}{P.lengthT(t')} = m' - \lfloor m' \rfloor = m - k - \lfloor m-k \rfloor$

The case $m < 0$ is analogous. ■

Example 3.3.16 (for the length and shift functions) *For the following partitioning P*

...	$[-100, 0[$	$[0, 100[$	$[100, 101[$	$[101, 500[$	...
...	-1	0	1	2	...

we have:

$P.lengthP(0, 100)$	=	1	$P.shiftP(0, 1)$	=	100
$P.lengthP(0, 50)$	=	0.5	$P.shiftP(0, 2)$	=	101
$P.lengthP(50, 101)$	=	1.5	$P.shiftP(50, 0.5)$	=	100
$P.lengthP(50, 200)$	=	1.7481203	$P.shiftP(100, -0.5)$	=	50
			$P.shiftP(50, 1.75)$	=	200 *(rounded)*
			$P.shiftP(200, -1.75)$	=	50 *(rounded)*

■

Unfortunately the $shiftP$ function yields in many cases results which are not what is intuitively expected. The following example illustrates the critical phenomenon: The month April has 30 days. Suppose the time point t is exactly in the middle of April (15th of April, midnight). If t is shifted with the $shiftP$ function by exactly 1 month, we do not end up at midnight in the middle of May, as one might expect. Since May has 31 days, the exact middle of May is at noon, the 15th of May. Therefore the $shiftP$ function moves in this case a time point from midnight to a time point at noon.

This phenomenon occurs always when the partitions have different lengths. A shift function, however, which can only take into account a single partitioning, has not much chance to do it better. If we expect to shift a time point t by one month from midnight, 15th of April to midnight, 15 of May, we take into account that months are decomposed into days. The shift is computed by realising that the time point t is 15 days away from the start of April. A shift of one month can then be realised by moving to a time point which is 15 days away from the start of May. PartLib provides a shift function which is based on a context of granularities and which can do the shift in this way (see Def. 3.3.36).

3.3.3 Labels Attached to Partitions

Labels can be attached to partitions in quite different ways. Therefore PartLib has an abstract interface for accessing the label information associated with partitions. The concrete algorithms depend on the way the labels are attached to the partitions. We start with the abstract interface and then describe the concrete methods.

Definition 3.3.17 (Label Attachments) *Given a partitioning $P = (CO, sPa)$, a* label attachment *is a, possibly partial, mapping $Label : CO.Co \mapsto$* `LABEL` *from partition coordinates to labels (Def. 3.2.2).*
Furthermore, for a time point t let $Label(t) \overset{\text{def}}{=} Label(P.PaCo(t))$. ■

Partitions have at most one label, but not all partitions need to have labels, and different partitions may have the same label. Partitions without label are called *gap partitions.*

Definition 3.3.18 (Labelled Partitionings) *A* Labelled Partitioning Representation *(or simply "labelled partitioning") is a tuple $P = (CO, sPa, Label)$ where (CO, sPa) is a Coordinate Based Portioning Representation (Def. 3.3.6) and $Label$ is a labelling attachment (Def. 3.3.17).*
Let $P.Label \overset{\text{def}}{=} Label$ be the associated label attachment. ■

There are a number of functions operating on labelled partitionings or granularities which require to locate the coordinate of a partition with a particular label. The search for such a coordinate may not terminate if the partitioning has no such label at all. Therefore it is important to have a function $P.hasLabel(l, LH)$ which *can decide* whether the labelled partitioning has such a label.

Definition 3.3.19 (A Label Existence Check)
A labelled Partitioning $P = (CO, sPa, Label)$ has a decidable "label existence check" for a label hierarchy LH if the functions

$$\begin{array}{lcll} P.hasLabel(l, LH) & \overset{\text{def}}{=} & \exists i \in CO.Co\ Label(i) \leq_{LH} l & \textit{and} \\ P.hasNoLabel(l, LH) & \overset{\text{def}}{=} & \neg\exists i \in CO.Co\ Label(i) \leq_{LH} l & \textit{and} \\ P.hasGap() & \overset{\text{def}}{=} & \exists i \in CO.Co\ Label(i) = \emptyset & \end{array}$$

are decidable for every label l and label hierarchy LH. ■

A labelled partitioning which is based on a finite labelling (Def. 3.3.21 below) has always a decidable label existence check. The functions need only check the finitely many labels in the finite labelling.

The functions in Def. 3.3.20 below can be used to locate partitions with given labels. They are partial functions if the given label is not one of the labels in the partitioning. Therefore they should be used only in combination with the *hasLabel* and *hasNoLabel* and *hasGap* functions (Def. 3.3.19).

Definition 3.3.20 (Navigators for Labellings)
Let $P = (CO, sPa, Label)$ be a labelled partitioning, $i \in CO.Co$ a partition coordinate, l a label and LH a label hierarchy.

$$\begin{array}{rcl} P.nextLabelled(i, l, LH, true) & \stackrel{\text{def}}{=} & min_{CO}\{j \geq i \mid P.Label(j) \leq_{LH} l\} \\ P.nextLabelled(i, l, LH, false) & \stackrel{\text{def}}{=} & max_{CO}\{j \leq i \mid P.Label(j) \leq_{LH} l\} \end{array}$$

nextLabelled moves forward (true case) or backwards (false case) to the coordinate of the next/previous partition whose label is below l in the label hierarchy. It is undefined if P has no such label.

$$\begin{array}{rcl} P.nextLabelled(i, true) & \stackrel{\text{def}}{=} & min_{CO}\{j \geq i \mid P.Label(j) \neq \emptyset\} \\ P.nextLabelled(i, false) & \stackrel{\text{def}}{=} & max_{CO}\{j \leq i \mid P.Label(j) \neq \emptyset\} \end{array}$$

nextLabelled without a label parameter moves forward (true case) or backwards (false case) to the coordinate of the next/previous partition with a non-empty label. It is undefined if P has no empty label.

$$\begin{array}{rcl} P.nextDiffererrnt(i, l, LH, true) & \stackrel{\text{def}}{=} & min_{CO}\{j \geq i \mid P.Label(j) \not\leq_{LH} l\} \\ P.nextDiffererrnt(i, l, LH, false) & \stackrel{\text{def}}{=} & max_{CO}\{j \leq i \mid P.Label(j) \not\leq_{LH} l\} \end{array}$$

$nextDifferent$ moves forward (true case) or backwards (false case) to the coordinate of the next/previous partition whose label is not *below l in the label hierarchy. It is undefined if P has only labels below or equal to l in the hierarchy.*

$$\begin{array}{rcl} P.nextGap(i, true) & \stackrel{\text{def}}{=} & min_{CO}\{j \geq i \mid P.Label(j) = \emptyset\} \\ P.nextGap(i, false) & \stackrel{\text{def}}{=} & max_{CO}\{j \leq i \mid P.Label(j) = \emptyset\} \end{array}$$

nextGap moves forward (true case) or backwards (false case) to the coordinate of the next/previous gap partition. It is undefined if P has no gap partition. ■

3.3.3.1 Finite Labellings

For many periodic temporal notions there are finitely many standard names for the partitions, which are repeated in regular intervals. For example, days are named 'Monday', 'Tuesday' etc., and repeated every 7th day. Months are named 'January', 'February' etc., and repeated every 12th month. Seasons are named 'winter', 'spring' etc. and repeated every 4th season. In these cases it is sufficient to list the finitely many labels, and to attach the first one in the list to the partition with coordinate 0. The labels for the other partitions can then be extrapolated to the past and the future.

Definition 3.3.21 (Labellings) *A* Labelling *L in PartLib is a finite sequence of labels $l_0, \ldots, l_{n-1}$. There can be multiple occurrences of the same label in the labelling and some of the labels l_i may be the empty label.*

A labelling $L = l_0, \ldots, l_{n-1}$ is turned into a labelling attachment *for a partitioning P and a coordinate i: $Label(i) \stackrel{\text{def}}{=} l_{i\%n}$.* ■

Example 3.3.22 (The Labelling of Days) *The origin of the reference time is again January 1^{st} 1970. This was a Thursday. Therefore we choose as labelling for the day partitioning*

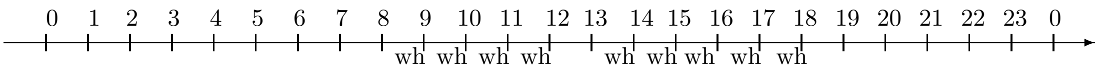
$$L \stackrel{\text{def}}{=} Th, Fr, Sa, Su, Mo, Tu, We.$$

The following correspondences are obtained:

$ref.time:$	$\dots$	$[-86400, 0[$	$[0, 86400[$	$[86400, 172800[$	$\dots$
$coordinate:$	$\dots$	-1	0	1	$\dots$
$label:$	$\dots$	We	Th	Fr	$\dots$

This means, for example, $Day.Label(-1) = We$, i.e. December 31 1969 was a Wednesday. ■

As we have seen in Sec. 3.2.1.3, partitionings with finite labellings are a very convenient basis for specifying various kinds of granularities. There is, however, a conceptual difficulty which needs special consideration. The difficulty can be illustrated with the working day example (Ex. 3.2.4).



A specification of the "my working day" granule could be: $partition = hour, label = wh$ with the implicit assumption that a granule consists of all wh-labelled partitions, possibly interrupted by gap partitions. The interval [8,18], with a gap at 12 is such a granule, but also, for example, [13,11 next day], which was not intended. To exclude such cases, we need a further assumption: "granules based on labelled partitionings with finite labellings do not exceed the boundaries of the labelling blocks". This condition rules out the interval [13,11 next day] in the above example.

The functions listed in the next definition (Def. 3.3.23) support the definition of "Labelled Granularities" (Def. 3.3.35) which rely on partitionings with finite labellings. LG stands for "Labelled Granularity". A labelled granule in the sense of Def. 3.3.23 with label l and label hierarchy LH is a maximally connected block of partitions which are either gap partitions or they are labelled with a label $l' \leq_{LH} l$, and which do not exceed labelling blocks. A labelling block for a labelling $(l_0, \dots, l_{n-1})$ is a block of partitions with coordinates $k, \dots, k = n$ for some k.

The functions in (Def. 3.3.23) are partial if no suitable granules exist. This is, however, easy to check because there are only the finitely many labels and coordinates of a labelling block to check.

Definition 3.3.23 (Navigators with Finite Labellings) *Let $P = (CO, sPa, Label)$ be a labelled partitioning, based on a finite labelling $(l_0, \dots, l_{n-1})$ (Def. 3.3.21), LH a label hierarchy, $i \in CO.Co$, a partition coordinate and l a label.*

$$\begin{aligned} P.withinLG(i, l, LH) \overset{\text{def}}{=} P.Label(i) \leq_{LH} l \vee \quad & (P.Label(i) = \emptyset \wedge \\ & P.Label(s) \leq_{LH} l \wedge \\ & P.Label(e) \leq_{LH} l \wedge \\ & \forall k\ s <_{CO} k \leq_{CO} e : k \,\%\, n \neq 0) \end{aligned}$$

where $s \overset{\text{def}}{=} P.nextLabelled(i, false)$ and $e \overset{\text{def}}{=} P.nextLabelled(i, true)$

checks whether the partition with coordinate i is within a labelled granule.

$$P.startLG(i, l, LH) \overset{\text{def}}{=} \begin{cases} min_{CO}\{j \leq_{CO} i \mid P.Label(j) \leq_{LH} l \wedge \forall k\ j <_{CO} k \leq_{CO} i : \\ \qquad (P.Label(k) = \emptyset \vee P.Label(k) \leq_{LH} l) \wedge \\ \qquad k \,\%\, n \neq 0\} & \textit{if } P.withinLG(i, l) \\ P.nextLabelled(i, l, LH, true) & \textit{otherwise} \end{cases}$$

computes the start coordinate of the next labelled granule. If i is within a granule then the start coordinate of this granule is returned. If i is not within a granule, then the start coordinate of the next (future) granule after i is computed. The function is partial if no such granule exists.

$$P.endLG(i,l,LH) \overset{\text{def}}{=} \begin{cases} max_{CO}\{j \geq_{CO} i \mid P.Label(j) \leq_{LH} l \wedge \forall k\ i \leq_{CO} k <_{CO} j : \\ \quad (P.Label(k) = \emptyset \vee P.Label(k) \leq_{LH} l) \wedge \\ \quad k \,\%\, n \neq 0\} & \text{if } P.withinLG(i,l,LH) \\ P.nextLabelled(i,l,LH,false) & \text{otherwise} \end{cases}$$

computes the end coordinate of the next labelled granule. If i is within a granule then the end coordinate of this granule is returned. If i is not within a granule, then the end coordinate of the previous (past) granule before i is computed. The function is partial if no such granule exists.

$$P.nextLG(i,l,LH,foward) \overset{\text{def}}{=} \begin{cases} (P.startLG(i,l,LH), P.endLG(i,l,LH)) \\ \quad \text{if } P.withinLG(i,l,LH) \\ (P.startLG(i,l,LH), P.endLG(P.startLG(i,l,LH),l,LH)) \\ \quad \text{if } \neg P.withinLG(i,l,LH) \wedge forward \\ (P.startLG(P.endLG(i,l,LH),l), P.endLG(i,l,LH)) \\ \quad \text{if } \neg P.withinLG(i,l,LH) \wedge \neg forward \end{cases}$$

computes start and end coordinates of the next labelled granule. If i is within a granule then the coordinates of this granule are returned. If i is not within a granule, then the end coordinate of the next ($forward = true$) or previous ($forward = false$) granule is computed. The function is partial if no such granule exists. ■

The **Labelling** class in PartLib has a sophisticated implementation of labellings where a lot of information is precomputed when a new labelling is defined. This way navigation along labelling sequences is very efficient and requires in many cases only a single table lookup.

3.3.4 Partitioning Based Granularity Representations

Granularities in PartLib represent sequences of non-overlapping granules, and granules are, possibly non-convex, time intervals. They are modelled at an abstract level which determines the interface to its general attributes and methods, and at a concrete level, which provides different ways to specify different kinds of granularities.

The basis of all granularities in PartLib are partitionings. That means, a particular granularity G is always associated with a partitioning P_G such that a granule $g \in G$ consists of one or more partitions from P_G. Thus, granules need not have their own coordinates to identify them, but they can be identified by the coordinates of the component partitions. The different kinds of granularities at the concrete level are distinguished by the different ways, partitions from P_G are comprised into granules.

Definition 3.3.24 (Partitioning Based Granularity Representations (PBGR))
A Partitioning Based Granularity Representations is a tuple

$$(P, enclosingGranule, withinGranule, firstGranule)$$

where

- $P = (CO, sPa, Label)$ *is the underlying labelled partitioning (Def. 3.3.18)*

- *enclosingGranule* : $CO.Co \mapsto CO.Co \times CO.Co$ *is a partial function for computing the boundaries of the granule around a partition coordinate. Thus,* $enclosingGranule(i) = (s, e)$ *where s and e are the coordinates of the start and end partition of the granule g around i, if there is one.*

- *withinGranule* : $CO.Co \mapsto bool$ *is a function which returns true if i is within a granule interval (and not in an internal gap), otherwise it returns false.*

- $firstGranule$: `Time` $\mapsto CO.Co \times CO.Co$ *is a partial function for computing the boundaries of the first granule after a time point t.* ■

The two functions *enclosingGranule* and *withinGranule* together specify granules. *enclosing-Granule* computes the boundary coordinates and *withinGranule* determines the internal (and external) gaps. Def. 3.3.35 contains a number of concrete definitions for these two functions.

The third function, $firstGranule$, has a very particular purpose. It determines *what counts* as the first granule after a given time point. An example where this is important is the first week within a year. If t is new year's eve then the first week in the new year is not necessarily the week around t, but the first week whose bigger part lies inside the new year. If new year's day is a Friday, Saturday or Sunday then this week is still taken to be in the last year. The $firstGranule$ function therefore becomes relevant when we want to count granules relative to some start time. The specification of a single $firstGranule$ function takes this phenomenon into account in a very general way. There could be cases where the first granule after a time point t is computed differently depending on the meaning of t. For example, somebody might determine the first week within a year differently to the first week within a month. If such cases really come up in particular applications, the $firstGranule$ function needs to get more parameters which allows it to distinguish the meaning of the time point parameter.

Definition 3.3.25 (Generated Granularities) *A Partitioning Based Granularity Representations* $G = (P, nextGranule, withinGranule, firstGranule)$ *generates the following granularity:*

$$\{ \bigcup_{\substack{s \leq j \leq e \\ withinGranule(j) = true}} [P.sPa(j), P.ePa(j)[\mid (s, e) = P.enclosingGranule(i), i \in CO.Co\} \quad ■$$

The granularity construction in (Def. 3.2.3) loops over all partition coordinates i for which $enclosingGranule(i)$ is defined to get the coordinates of the boundary partitions of the granule around i. All partitions between the boundary partitions, for which $withinGranule(j)$ yields true are then comprised into one granule.

Proposition 3.3.26 *The generated granularity in Def. 3.3.25 is really a granularity.*

Proof: This holds:

1. by the very definition of a granule (Def. 3.2.3) and because *enclosingGranule* computes the coordinates of the first and last partition of a granule and

2. the construction in Def. 3.3.25 uses the *withinGranule* function to exclude internal gaps.

■

All granule functions which take partition coordinates as parameters can easily be extended to take time points as parameters: the *PaCo* function of the underlying partitioning turns the time point into the corresponding partition coordinate, and then the original granule function can be called.

Definition 3.3.27 (Granule Functions with Time Parameters) *For a Partitioning Based Granularity Representations $G = (P, enclosingGranule, withinGranule, firstGranule)$ (Def. 3.3.24) and any n-place function $G.F$ of type*

$$G.F : X_0 \times \ldots \times X_k \times CO.Co \times X_{k+2} \times \ldots \times X_{n-1} \mapsto Y_0 \times \ldots \times Y_m$$

we define for parameters x_i of type X_i and time point t:

$$G.F(x_0, \ldots, x_k, t, x_{k+2}, \ldots, x_{n-1}) \overset{\text{def}}{=} G.F(x_0, \ldots, x_k, P.PaCo(t), x_{k+2}, \ldots, x_{n-1})$$

■

Sometimes it is useful to check whether a partition coordinate i lies between the boundaries of a granule, regardless whether it is an internal gap partition or not.

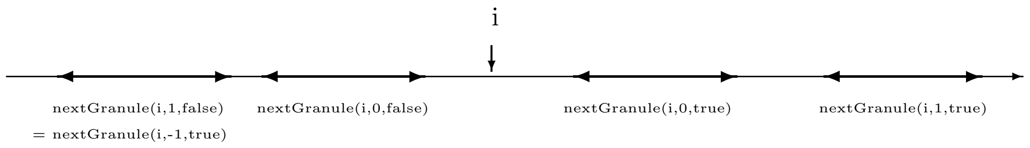
Definition 3.3.28 (within Granule Boundaries) *For a Partitioning Based Granularity Representations $G = (P, enclosingGranule, withinGranule, firstGranule)$ and a partition coordinate i we define*

$$withinGrBoundaries(i) \overset{\text{def}}{=} \begin{cases} s \leq_{CO} i \leq_{CO} e & \textit{if}(s,e) \overset{\text{def}}{=} enclosingGranule(i) \textit{ is defined} \\ false & \textit{otherwise} \end{cases}$$

■

An important feature of the concept of granularities is the possibility to move from a given granule forward or backward to the next or previous granule. In Def. 3.3.29 below we therefore define a *nextGranule* function. $nextGranule(i, forward)$ computes either the granule boundaries of the granule around a partition coordinate i, if there is one, or the next/previous granule after/before i, depending on the parameter $forward$.

The second version, $nextGranule(i, m, forward)$ yields the m'th next/previous granule. If i is within a granule, them $m = 0$ yields this granule. $m = 1$ yields the next/previous granule, depending on the value of $forward$. If i is outside a granule, $m = 0$ yields the next/previous granule after/before i, $m = 1$ jumps two granules further etc.

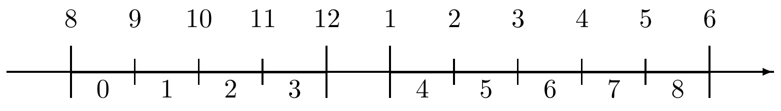


Definition 3.3.29 (nextGranule) *For a Partitioning Based Granularity Representations $G = (P, enclosingGranule, withinGranule, firstGranule)$, a partition coordinate i and a Boolean parameter $forward$ we define a partial function $nextGranule : CO.Co \times Bool \mapsto CO.Co \times CO$:*

$$G.nextGranule(i, forward) \overset{\text{def}}{=} \begin{cases} (s,e) = G.enclosingGranule(i) \text{ if this is defined} \\ min_j\{(s,e) \mid j > i \wedge (s,e) = G.enclosingGranule(j) \text{ is defined} \mid j\} \\ \quad \text{if } forward \\ max_j\{(s,e) \mid j < i \wedge (s,e) = G.enclosingGranule(j) \text{ is defined} \mid j\} \\ \quad \text{if } \neg forward \end{cases}$$

■

The *nthGranule* function (Def. 3.3.30 below) below is a generalisation of the *nextGranule* function (Def. 3.3.29) in two ways. First of all, it can move n granules forward or backwards where n is an arbitrary integer. Secondly, it can modify the counting of the granules by a second granularity. The purpose of this feature should become clear when we look at the working day example again (Ex. 3.2.4).



Two granularities are involved here, the hour granularity G_{hour} and the working day granularity G_{wd}. If it is 8 AM and we want to locate, say, the 5th hour within the working day, there are two possibilities.

1. The lunch break counts as well as an hour within the working day. The 5th hour within the working day is then between 1 PM and 2 PM. That means, the structure of the G_{wd} is irrelevant.

2. The lunch break does not count. The 5th hour is then between 2 PM and 3 PM.

The first possibility is realised by calling the *nthGranule* function:

$$G_{hour}.nthGranule(i, n, forward, \emptyset),$$

i.e. the second granularity is undefined. The second possibility is realised by the call

$$G_{hour}.nthGranule(i, n, forward, G_{wd}).$$

The Boolean parameter $forward$ is relevant only if $n = 0$ and the partition coordinate i is within an (internal or external) gap of the G or G'-granularities. $forward = true$ causes to search forward to find the first partition coordinate which is within G-granule and not within a G'-gap. $forward = false$ causes a backwards search.

Definition 3.3.30 (nthGranule) *For Partitioning Based Granularity Representations G and G', a partition coordinate i, an integer n and a Boolean parameter $forward$ we define a partial function $nthGranule : CO.Co \times \mathbb{I} \times Bool \times Granularities \mapsto CO.Co \times CO.Co$*

$$G.nthGranule(i,n,forward,G') \overset{\text{def}}{=} \begin{cases} G.nextGranule(i', forward) & \textit{if } n = 0 \\ G.nthGranule(i, -n, \neg forward, G') & \textit{if } \neg forward \\ G.nthGranule(e +_{G.CO} 1, n-1, true, G') & \textit{if } n > 0 \\ \quad \textit{where } (s,e) = G.nextGranule(i, true) & \\ G.nthGranule(s -_{G.CO} 1, n+1, false, G') & \textit{if } n < 0 \\ \quad \textit{where } (s,e) = G.nextGranule(i, false) & \end{cases}$$

where i' is defined as follows if $G' \neq \emptyset$:

$$i' \overset{\text{def}}{=} \begin{cases} min\{j \geq_{G.CO} i \mid G'.withinGranule(P.sPa(j)) \wedge G.withinGrBoundaries(j)\} & \textit{if } forward \\ max\{j \leq_{G.CO} i \mid G'.withinGranule(P.sPa(j)) \wedge G.withinGrBoundaries(j)\} & \textit{otherwise} \end{cases}$$

and i' is defined as follows if $G' = \emptyset$:

$$i' \overset{\text{def}}{=} \begin{cases} min\{j \geq_{G.CO} i \mid G.withinGrBoundaries(j)\} & \textit{if } forward \\ max\{j \leq_{G.CO} i \mid G.withinGrBoundaries(j)\} & \textit{otherwise} \end{cases}$$

furthermore, $P \overset{\text{def}}{=} G.partitioning$

Let $G.nthGranuleI(i,n,forward,G') \overset{\text{def}}{=} [P.sPa(s), P.ePa(e)[$
where $(s,e) \overset{\text{def}}{=} G.nthGranule(i,n,forward,G')$
(nthGranuleI computes the granule boundaries as time interval.)

Let $G.nthGranule(i,n,forward) \overset{\text{def}}{=} G.nthGranule(i,n,forward,\emptyset)$ and
$G.nthGranuleI(i,n,forward) \overset{\text{def}}{=} G.nthGranuleI(i,n,forward,\emptyset)$. ■

Since the granules in PartLib are based on partitionings, we can measure the length of a granule either in terms of the basic time units or in terms of the number of partitions which make up a granule. The function *lengthT*(i) in Def. 3.3.31 below measures the length of a granule around a partition coordinate i in terms of the basic time units. The function *lengthP*(i) measures the length of a granule around a partition coordinate i in terms of the number of partitions. Both functions do not measure internal gaps and they return 0 if i lies outside a granule.

Definition 3.3.31 (Length of Granules) *For a Partitioning Based Granularity Representations $G = (P, enclosingGranule, withinGranule, firstGranule)$, we define functions $lengthT : CO.Co \mapsto \texttt{Time}$ and $lengthP : CO.Co \mapsto \mathbb{N}$. Let i be a partition coordinate.*

$$G.lengthT(i) \overset{\text{def}}{=} \begin{cases} 0 & \textit{if } \neg withinGrBoundaries(i) \\ \sum_{\substack{s \leq_{CO} j \leq_{CO} e \\ withinGranule(j)}} P.lengthT(j) & \textit{otherwise} \\ & \textit{where } (s,e) \overset{\text{def}}{=} enclosingGranule(i) \end{cases}$$

computes the length of a granule (without internal gaps) in terms of the basic time units.

$$G.lengthP(i) \overset{\text{def}}{=} \begin{cases} 0 & \textit{if } \neg withinGrBoundaries(i) \\ \sum_{\substack{s \leq_{CO} j \leq_{CO} e \\ withinGranule(j)}} 1 & \textit{otherwise} \\ & \textit{where } (s,e) \overset{\text{def}}{=} enclosingGranule(i) \end{cases}$$

computes the length of a granule (without internal gaps) in terms of the number of partitions. ■

The three length functions in Def. 3.3.32 below measure distances between time points in terms of 1. the basic time units, 2. a, possibly fractional, number of partitions, and 3. a, possibly fractional, number of granules. The difference to the corresponding length functions for partitionings (Def. 3.3.13) is that internal and external gaps are ignored.

Definition 3.3.32 (Length Functions for Granules) *We define three functions which measure the distance between two time points while ignoring the gaps between and inside granules.*

1. $G.lengthT : \texttt{Time} \times \texttt{Time} \mapsto \texttt{Time}$ *measures distances in terms of time length*
2. $G.lengthP : \texttt{Time} \times \texttt{Time} \mapsto \mathbb{R}$ *measures distances in terms of numbers of partitions (maybe fractional)*
3. $G.lengthG : \texttt{Time} \times \texttt{Time} \mapsto \mathbb{R}$ *measures distances in terms of numbers of granules (maybe fractional)*

For a Partitioning Based Granularity Representations

$$G = (P, enclosingGranule, withinGranule, firstGranule)$$

and two time points t_0 and t_1 let

$$\begin{array}{lcl}(s_0, e_0) & \stackrel{\text{def}}{=} & nextGranule(t_0, true) \\ (s_1, e_1) & \stackrel{\text{def}}{=} & nextGranule(t_1, false).\end{array}$$

All three functions below yield 0 if $nextGranule(t_0, true)$ or $nextGranule(t_1, false)$ is undefined. Suppose now that s_0, e_0, s_1 and e_1 are defined.

If $t_0 \leq t_1$ we define

$$G.lengthT(t_0, t_1) \stackrel{\text{def}}{=} \sum_{\substack{s_0 \leq_{CO} i \leq_{CO} e_1 \\ G.withinGranule(i)}} \begin{cases} 0 & \text{if } P.ePa(i) \leq t_0 \vee t_1 \leq P.sPa(i) \\ P.ePa(i) - t_0 & \text{if } P.sPa(i) \leq t_0 < P.ePa(i) \\ t_1 - P.sPa(i) & \text{if } P.sPa(i) \leq t_1 < P.ePa(i) \\ P.lengthT(i) & \text{otherwise} \end{cases}$$

If $t_0 > t_1$ we define $G.lengthT(t_0, t_1) \stackrel{\text{def}}{=} -G.lengthT(t_1, t_0)$.

If $t_0 \leq t_1$ we define

$$G.lengthP(t_0, t_1) \stackrel{\text{def}}{=} \sum_{\substack{s_0 \leq_{CO} i \leq_{CO} e_1 \\ G.withinGranule(i)}} \begin{cases} 0 & \text{if } P.ePa(i) \leq t_0 \vee t_1 \leq P.sPa(i) \\ \frac{P.ePa(i) - t_0}{P.lengthT(i)} & \text{if } P.sPa(i) \leq t_0 < P.ePa(i) \\ \frac{t_1 - P.sPa(i)}{P.lengthT(i)} & \text{if } P.sPa(i) \leq t_1 < P.ePa(i) \\ 1 & \text{otherwise} \end{cases}$$

If $t_0 > t_1$ we define $G.lengthP(t_0, t_1) \stackrel{\text{def}}{=} -G.lengthP(t_1, t_0)$.

If $t_0 \leq t_1$ and initially $(s, e) \stackrel{\text{def}}{=} (s_0, e_0)$ we define
$G.lengthG(t_0, t_1) \stackrel{\text{def}}{=}$

$$\sum_{\substack{(s,e) \stackrel{\text{def}}{=} G.nextGranule(e +_{CO} 1, true) \\ e \leq_{CO} e_1}} \begin{cases} \frac{G.lengthT(t_0, P.ePa(e))}{G.lengthT(s)} & \text{if } P.sPa(s) \leq t_0 < P.ePa(e) \\ \frac{G.lengthT(P.sPa(s), t)}{G.lengthT(s)} & \text{if } P.sPa(s) \leq t_1 < P.ePa(e) \\ 1 & \text{otherwise} \end{cases}$$

If $t_0 > t_1$ we define $G.lengthG(t_0, t_1) \stackrel{\text{def}}{=} -G.lengthG(t_1, t_0)$.

■

The function granulesWithLabels (Def. 3.3.33 below) computes for a time interval $[t_0, t_1[$ or a partition coordinate interval $[a, b[$ a list of relative granule positions of granules between t_0 (inclusive) and t_1 (exclusive) or between a (inclusive) and b (exclusive) respectively which have at least one partition whose label is in a given list L of labels, or which is a sub-label of some label in this list. The function is used in Def. 3.3.79 and Def. 3.3.80.

Definition 3.3.33 (granulesWithLabels) *For two granularities G and G', a time interval $[t_0, t_1[$ a set L of labels and a label hierarchy LH we define*

$$\begin{aligned}&G.granulesWithLabel(t_0, t_1, L, LH, G') \stackrel{\text{def}}{=}\\ &\quad \{i \in \mathbb{N} \mid (s, e) = G.nthGranule(s_0, i, true, G') \wedge e <_{G.CO} s_1 \wedge\\ &\qquad \exists j \; s \leq_{G.CO} j \leq_{G.CO} e \wedge P.Label(j) \leq_{LH} l \textit{ for some } l \in L\}\end{aligned}$$

where
1. $(s_0, e_0) \stackrel{\text{def}}{=} G.firstGranule(t_0)$
2. $(s_1, e_1) \stackrel{\text{def}}{=} G.firstGranule(t_1)$
3. $P \stackrel{\text{def}}{=} G.partitioning.$

For two G-partition coordinates a, b with $a \leq_{G.CO} b$ we define

$$\begin{aligned}&G.granulesWithLabel(a, b, L, LH, G') \stackrel{\text{def}}{=}\\ &\quad \{i \in \mathbb{N} \mid (s, e) = G.nthGranule(a, i, true, G') \wedge e <_{G.CO} b \wedge\\ &\qquad \exists j \; s \leq_{G.CO} j \leq_{G.CO} e \wedge P.Label(j) \leq_{LH} l \textit{ for some } l \in L\}\end{aligned}$$

■

A number of useful auxiliary functions are listed in the following definition.

Definition 3.3.34 (Further Auxiliary Functions) *For a granularity G and a time point t we define*

$$\begin{aligned}G.withinInternalGap(t) &\stackrel{\text{def}}{=} G.withinGrBoundaries(t) \wedge \neg G.withinGranule(t)\\ G.withinExternalGap(t) &\stackrel{\text{def}}{=} \neg G.withinGrBoundaries(t)\end{aligned}$$

Let $G.internalGapGranularity()$ be the granularity whose granules consist of G's internal gap partitions.
Let $G.externalGapGranularity()$ be the granularity whose granules consist of G's external gap partitions.
Both functions are undefined if there are no such gaps. ■

PartLib distinguishes different types of granularities according to the different ways they can be constructed from labelled partitionings.

Partitioning Granularities just identify partitions and granules. Thus, all partitionings are automatically associated with a granularity. Typical examples, are years, months, weeks, days, hours, minutes, seconds.

Label Partition Granularities for a label l in a label hierarchy LH choose individual partitions labelled with l or a sub-label of l in LH. As an example, consider the 'day' partitioning with labels Mo, Tu, We, Th, Fr, Sa, Su. A Label Partition granularity with label We in a flat label hierarchy selects all Wednesdays as granules. The other days are the gaps between the granules.

If the label hierarchy LH classifies the labels Su and So as 'weekend day', we could define a Label Partition Granularity with label 'weekend day'. This way, we get two consecutive granules, the Saturday and the Sunday, followed by a gap up to the next Friday.

Label Block Granularities for a label l in a label hierarchy LH are very similar to Label Partition Granularity. The difference is that neighbouring Label Partition granules are comprised into one single Label Block granule. Taking the above example again, we could define a Label Block Granularity with label 'weekend day'. It joins the Saturday and Sunday into one single granule.

Labelling Granularities for a label l in a label hierarchy LH are defined for labelled partitionings with finite labellings (Def. 3.3.21) only. They are essentially like Label Block Granularities, but the granules do not exceed the boundaries of the finite labellings. The example before Def. 3.3.23 explains the ideas.

Gap Block Granularities finally, comprise neighbouring unlabelled partitions into a single granule

Definition 3.3.35 (Concrete Granularities)
A granularity (P,nextGanule,withinGranule,firstGranule) with $P = (CO, sPa, Label)$ *is a*

Partitioning Granularities *iff the 'enclosingGranule' function identifies partitions of P and granules, i.e. granules are just partitions:*

$$\begin{array}{lcl} \mathit{enclosingGranule}(i) & \overset{\text{def}}{=} & (i,i) \\ \mathit{withinGranule}(i) & \overset{\text{def}}{=} & \mathit{true} \end{array}$$

Label Partition Granularities *with label l and label hierarchy LH iff P is a labelled partitioning, and the 'enclosingGranule' function chooses partitions labelled with l or a sub-label of l in LH as granule, i.e. granules are individual partitions labelled l (or sub-labels of l)*

$$\begin{array}{lcl} \mathit{enclosingGranule}(i) & \overset{\text{def}}{=} & \begin{cases} (i,i) & \text{if } P.Label(i) \leq_{LH} l \\ undefined & \text{otherwise} \end{cases} \\ \mathit{withinGranule}(i) & \overset{\text{def}}{=} & \begin{cases} true & \text{if } P.Label(i) \leq_{LH} l \\ false & \text{otherwise} \end{cases} \end{array}$$

Label Block Granularities *with label l and label hierarchy LH iff P is a labelled partitioning, and the 'enclosingGranule' function chooses consecutive blocks of partitions labelled with l or a sub-label of l in LH as granule, i.e. granules are partition blocks labelled l (or sub-labels of l):*

$$\mathit{enclosingGranule}(i) \overset{\text{def}}{=}$$
$$\begin{cases} (P.nextDiffererent(i, l, LH, false) +_{CO} 1, P.nextDiffererent(i, l, LH, true) -_{CO} 1) \\ \quad \text{if } P.Label(i) \leq_{LH} l \qquad \text{(Def. 3.3.20)} \\ undefined \quad \text{otherwise} \end{cases}$$

$$\mathit{withinGranule}(i) \overset{\text{def}}{=} \begin{cases} true & \text{if } P.Label(i) \leq_{LH} l \\ false & \text{otherwise} \end{cases}$$

Labelled Granularities *with label l and label hierarchy LH iff P is a labelled partitioning with finite labellings (Def. 3.3.21), and the 'enclosingGranule' function chooses partitions labelled with l or a sub-label of l in LH as granule in the sense of Def. 3.3.23:*

$$enclosingGranule(i) \overset{\text{def}}{=} \begin{cases} P.nextLG(i, l, LH, true) & \text{if } P.withinLG(i, l, LH) \\ undefined & otherwise \end{cases}$$

$$withinGranule(i) \overset{\text{def}}{=} P.withinLG(i, l, LH) \qquad \text{(Def. 3.3.23)}$$

Gap Block Granularities *iff P is a labelled partitioning, and the 'enclosingGranule' function chooses consecutive sequences of unlabelled partitions as granules, i.e.*

$$enclosingGranule(i) \overset{\text{def}}{=} \begin{cases} (P.nextLabelled(i, false) +_{CO} 1, P.nextLabelled(i, true) -_{CO} 1) \\ \quad \text{if } P.Label(i) = \emptyset \\ undefined \quad otherwise \end{cases}$$

$$withinGranule(i) \overset{\text{def}}{=} \begin{cases} true & \text{if } P.Label(i) = \emptyset \\ false & otherwise \end{cases}$$

■

3.3.5 Granularity Based Time Shifts

Linguistic notions like "day" can denote partitionings and they can be used for representing distances between time points. There is a third meaning which is unfortunately different to the two other ones. If I say, for example, "let us meet again in 3 days", this means a *time shift.* The current point in time is shifted by a certain interval.

It turns out that the concept of time shifts is absolutely nontrivial. The following examples illustrate the phenomena we need to deal with:

- If it is January 5th, 8:53:10 AM, what does "one month from now" mean? Does it mean February, 5th, 8:53:10 AM, or just February 5th 12 AM, or February, 5th 8:00 AM because it is the start of a working day?
- If it is January 20th, what does "1.5 month from now" mean? "One month from now" obviously means February 20th, but what about the remaining 0.5 month? Does it refer to the month length of January, February or March, or even a combination of them?
- Suppose it is January 30th. Since there is no February 30th, what does "one month from now" mean?
- Let us consider granularities now. Take a granularity *working day* (Ex. 3.2.4) with granules lasting from 8 AM until 6 PM.
  - If it is 9 AM, "one working day from now" could mean 9 AM tomorrow, but also 8 AM tomorrow.
  - If it is 9 AM, "0.5. working days from now" obviously means 2 PM, but what does "0.5 working days from now" mean when it is 3 PM or, even worse, 10 PM, i.e. outside the working hours.

- Consider now a working day consisting of a day shift from 8 AM until 5 PM with a one hour lunch break at noon, and a night shift from 6 PM until 8 AM the next day (without break). What could "0.5 working days from now" mean when it is 9 AM, 12:30 PM, 7 PM or midnight?

One can easily find more and more complex examples where it becomes less and less clear what the desired meaning should be. The transition between "it is absolutely clear what the desired result should be" and "there is no idea at all what it means" is gradual. PartLib takes this into account by offering a very flexible shift function which can be parameterised in various ways to deal with many of the above mentioned phenomena.

A first idea for defining a granularity based time shift consists of the following steps:

1. We introduce a *date format*, say, year/month/day hour:minute:second.
2. We represent a time point t as a date in the given date format, for example, 2008/5/3 12:25:39
3. A time shift of, say, 1 month, could be realised by incrementing the month date by 1: 2008/6/3 12:25:39.

A time shift by a fractional value, say 2.5 month, causes an iteration over the date format.

1. We increment the month date by 2: 2008/7/3 12:25:39. A further shift by 0.5 month is necessary.
2. July has 31 days. $0.5 \cdot 31 = 15.5$. In the next step we therefore perform a shift by 15 days: 2008/7/18 12:25:39. A further shift by 0.5 days is necessary.
3. One day has 24 hours. $0.5 \cdot 24 = 12$. We shift by 12 hours: 2008/7/18 24:25:39 which could be normalised to 2008/7/19 0:25:39.

There are a number of refinements of this basic procedure

Limited Shift Level: One can restrict the iteration through the date format. For example, a shift of 2008/5/3 12:25:39 by 1 month up the *level of days* yields 2008/5/3 0:0:0. A shift by 2.5 months *up to the month level* yields 2008/7.

A suitable positive integer `level` (level 0 = year, level 1 = month etc.) is sufficient to stop the iteration.

Squeezed Shift: The basic procedure may produce unwanted results if the partitions have different lengths. Consider the date 2007/1/31 (31st of January 2007) The result of a 1 month shift would be 2007/2/31 (3rd of March 2007), which might not be the intended result. One may want to *squeeze* the shifted time point into February.

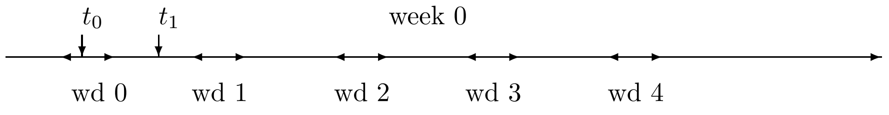
The 31st of January is the last day in January. A shift by 1 month could be a jump to the last day in February. But what about the second but last day in January or the third but last day?

A possible algorithmic approach to this problem could be as follows: let l_0 be the length the "source month" in terms of days and let l_1 be the length of the "target month", also in terms of days. For January it is $l_0 = 31$ and for February it is $l_1 = 28$. The 31st of January is 30 days away from the beginning of January. This yields a fraction 30/31 which can be multiplied with the length of February: $(30/31) \cdot 28 = 27.09$ The *rounded number* 27 is then added to the beginning of February and we get in fact 2007/2/28 as the shifted date. The procedure shows the following behaviour for a shift from January to February (in non-leap years): the days 1-6 are shifted to the same day in February. The days 7-16 are shifted by -1 day in February. The days 17-26 are shifted by -2 days and the days 27-31 are shifted by -3 days.

This procedure, which *squeezes* a time point which lies in a "source granule" to a corresponding time point in the "target granule", is activated by the Integer parameter $jumpPosition < 0$ in the $shiftG$ function in Def. 3.3.36.

Granularities with External Gaps: We come back to the working day example (Ex. 3.2.4). Suppose a working day is between 8 AM and 6 PM (without lunch break), from Monday to Friday, and we want to shift by "working days". A suitable date format could be

$$week/working\_day\ hour : minute : second$$



Suppose a time point t_0 yields the date 0/1 3:10:20 (week 0, first working day in this week, 3:10:20 hours away from the start of this working day, i.e. Monday, 11:10:20 AM.). A shift by 1 working day yields 0/1 3:10:20. A shift by 1.8 working days could be processed as follows:

1. A shift by 1 working day yields 0/1 3:10:20. There is a remaining shift of 0.8 working days.
2. The length of the granule is 8 hours. $0.8 \cdot 8 = 6.4$ hours. $3 + 6 = 9$, which is 1 hour beyond the granule. Thus, we move 1 granule forward to 0:2 0:10:20. The remaining shift of 1 hour yields 0:2 1:10:20. There is a remaining shift of 0.4 hours.
3. 0.4 hours is 24 minutes and the final result of the shift is therefore 0/2 1:34:20 (week 0, Wednesday, 9:34:20 AM).

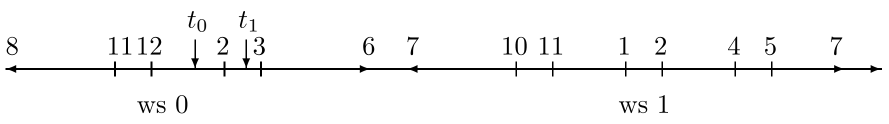
As a second example, take the time point t_1 which lies in the gap between working day 0 and working day 1. There are now two different ways to deal with the fact that t_1 lies in a gap between granules.

1. We could ignore external gaps completely and move t_1 in a preliminary step to the start of working day 1. A further shift by 1 working day would end up at the start of working day 2. This could be a reasonable interpretation of "let us wait for 1 working day".
2. We switch from the 'working day' granularity to the corresponding external gap block granularity (Def. 3.3.34) where the granules consist of the external gaps. The corresponding date format for the shift function would be now

   $$week/working\_day\_gaps\ hour : minute : second.$$

   This version is sensitive to the structure of the internal gaps.

Granularities with Internal Gaps: The situation is even more complicated when the granules have internal gaps. In the next example we have therefore a granularity 'working shifts' with a day shifts and night shifts. A day shift granule lasts, say from 8 AM - 6 PM, with one hour breaks at 11 AM and 2 PM. The night shift granule lasts from 7 PM - 7 AM the next day, with one hour breaks at 10 PM, 1 AM and 4 AM.



The date format is now

$$week/work\ shift/hour/minute/second.$$

A time point t_0 which lies in day shift ws 0, and not in an internal gap can be shifted by computing its distance to the start of the granule, while ignoring internal gaps.

Example: the date is 0/0 4:10:20 (week 0, ws 0, 4:10:20 hours away from the start of ws 0, i.e. 1:10:20 PM). A shift by 1 work shift jumps from ws 0 to ws 1 and there 4:10:20 hours away from the start of ws 1. We end up at 12:10:20 AM.

Now consider a time point t_1 which lies in an internal gap. There are again 2 possibilities.

1. Internal gaps are ignored completely. Therefore t_1 is moved to the start of the next non-gap part of the granule before the main shift operation is performed.
2. We switch from the working shift granularity to the corresponding external gap block granularity (Def. 3.3.34) where a granule consists of the internal gaps of the working shift granularity. This granularity has also internal gaps, but t_1 lies in a non-gap part of the gap block granules. It can therefore be treated like t_0 above.

For the granularity based time shift (Def. 3.3.36 below), we assume that each granularity G has three additional attributes.

1. The first attribute is a *sub–granularity* (*G.subGranularity*). This is a way to define date formats. For example, the year granularity can have the month granularity as sub–granularity, the month granularity can have the day granularity as sub–granularity etc. This yields a date format year/month/day/hour/minute/second. If a granularity has no sub–granularity any more then the $shiftG$ function below stops the recursion. PartLib assigns default sub–granularities. They can be overridden by the user.

   The mechanism with sub–granularities, however, makes it a bit cumbersome to change date formats. For example, a user might want to do a first shift operation with a date format year/month/day, followed by a shift with a date format year/week/day. In order to do this, it is necessary to exchange the sub–granularity of the year granularity.

   The advantage of this mechanism is that the $shiftG$ operation need not have too many extra control parameters.

2. The second attribute is the jump position (*G.jumpPosition*), which is -1 by default. The *jumpPosition* attribute defines the position from which sub–granules within granules are counted backwards. For example, the jump position for the month granularity (with the day granularity as sub–granularity) may be 25. This means that from day 25 onwards, a shift of one month is counted backwards. For example, day 24 in January (counting from day 0) is shifted to day 24 in February, whereas day 25 in January (which is 6 days before the end of January) is shifted to day 23 in February (6 days before the end of February n non-leap years). If $jumpPosition = -1$ then the shifts are 'squeezed' into the target directory (see the explanation above).

3. Finally the third attribute is the Boolean parameter *useGapGranularity*. If it is true then shifts from positions inside gaps use the corresponding internal or external gap granularity (Def. 3.3.34) instead of the original granularity. If it is false, then time points inside gaps are moved forward/backwards (depending on whether it is a forward or backward shift) to the next non-gap position before the actual shift is performed.

Definition 3.3.36 (A Time Shift Function for Granularities)
Let G be a granularity, t a time point, $m \in \mathbb{R}$, and $level \in \mathbb{N} \cup \{-1\}$. ($m$ is the amount of G-granules to be shifted forward (if $m > 0$) or backwards (if $m < 0$) level controls the depth of the recursion. $level = -1$ means recursion until there is no sub–granularity any more.)
We define a time shift function for granularities:

$$G.shiftG(t, m, level) \stackrel{\text{def}}{=} \begin{cases} t & \textit{if } m = 0 \\ G'.shiftGRec(t', s_0, e_0, s_1, e_1, m - \lfloor m \rfloor, level) & \textit{otherwise} \end{cases}$$

where

1. **If** $G.withinExternalGap(t)$ **then**

- **If** $G.useGapGranularity = true$ **then**
  $G' \stackrel{\text{def}}{=} G.externalGapGranularity()$ *and* *(Def. 3.3.34)*
  $t' \stackrel{\text{def}}{=} t$
- **else**
  $G' \stackrel{\text{def}}{=} G$ *and*
  t' is the start of the next granule if $m > 0$ and there is a next granule (otherwise it is the end of the previous granule -1) and
  t' is the end of the previous granule -1, if $m < 0$ and there is a previous granule (otherwise it is the start of the next granule).

2. **If** $G.withinInternalGap(t)$ **then**

- **If** $G.useGapGranularity = true$ **then**
  $G' \stackrel{\text{def}}{=} G.interalGapGranularity()$ *and* *(Def. 3.3.34)*
  $t' \stackrel{\text{def}}{=} t$
- **else**
  $G' \stackrel{\text{def}}{=} G$ *and*
  t' is the start of the next non-gap partition if $m > 0$ and
  t' is the end of the previous non-gap partition -1, if $m < 0$.

3. **Otherwise** $G' \stackrel{\text{def}}{=} G$ *and* $t' \stackrel{\text{def}}{=} t$.

4. $(s_0, e_0) \stackrel{\text{def}}{=} G'.enclosingGranule(t')$ *and*
   $(s_1, e_1) \stackrel{\text{def}}{=} G'.nextGranule(s_0, \lfloor m \rfloor, true)$

and $G.shiftGRec(t, s_0, e_0, s_1, e_1, m, level)$ *is defined recursively as follows:*
Let $P \stackrel{\text{def}}{=} G.partitinoing$ *and* $G' \stackrel{\text{def}}{=} G.subGranularity$.
If $G' = \emptyset$ *then* $G.shiftGRec(t, s_0, e_0, s_1, e_1, m, level) \stackrel{\text{def}}{=} P.sPa(s_1)$.

Now we assume $G' \neq \emptyset$

1. **If** $G'.withinExternalGap(t)$ **then**
   - **If** $G'.useGapGranularity = true$ **then**
     $G' := G'.externalGapGranularity()$ *and*
     $t' \stackrel{\text{def}}{=} t$
   - **else**
     t' is the start of the next G'-granule if $m > 0$ and there is a next granule (otherwise it is the end of the previous granule -1) and
     t' is the end of the previous G'-granule -1, if $m < 0$ and there is a previous granule (otherwise it is the start of the next granule).
2. **If** $G'.withinInternalGap(t)$ **then**
   - **If** $G'.useGapGranularity = true$ **then**
     $G' := G.interalGapGranularity()$ *and*
     $t' \stackrel{\text{def}}{=} t$
   - **else**
     t' is the start of the next non-gap partition if $m > 0$ and
     t' is the end of the previous non-gap partition -1, if $m < 0$.
3. **Otherwise** $t' \stackrel{\text{def}}{=} t$.
4. $l_0 \stackrel{\text{def}}{=} G'.lengthG(P.sPa(s_0), P.ePa(e_0))$ *(Def. 3.3.32)*
   $l_1 \stackrel{\text{def}}{=} G'.lengthG(P.sPa(s_1), P.ePa(e_1))$
5. $d \stackrel{\text{def}}{=} G'.lengthG(P.sPa(s_0), t)$
6. $$m' \stackrel{\text{def}}{=} m \cdot l_1 + \begin{cases} \lfloor d \cdot l_1 / l_0 \rfloor & \textit{if } G.jumpPosition < 0 \\ d & \textit{if } d \leq G.jumpPosition \vee l_0 = l_1 \\ l_1 - (l_0 - d) & \textit{if } l_0 < l_1 \\ \quad max(jumpPos, l_0 - (l_1 - d)) & \textit{otherwise} \end{cases}$$
7. *If the remaining shift m' exceeds the G-granule boundaries we need to move forward/backwards to the next G-granule.*

```
while($m' \geq l_1$)
        $(s_1, e_1) := G.nextGranule(e_1 +_{CO} 1, true)$
        $m' := m' - l_1$
        $l_1 := G'.lengthG(P.sPa(s_1), P.ePa(e_1))$

while($m' < 0$)
        $(s_1, e_1) := G.nextGranule(s_1 -_{CO} 1, false)$
        $m' := m' + G'.lengthG(P.sPa(s_1), P.ePa(e_1))$
```

8. $(s'_0, e'_0) \overset{\text{def}}{=} G'.enclosingGranule(t)$
$(s'_1, e'_1) \overset{\text{def}}{=} G'.nextGranule(P.sPa(s_1), \lfloor m' \rfloor, true)$

$G.shiftGRec(t, s_0, e_0, s_1, e_1, m, level) \overset{\text{def}}{=} G'.shiftGRec(t', s'_0, e'_0, s'_1, e'_1, m' - \lfloor m' \rfloor, level - 1).$ ■

3.3.6 Durations

The *lengthG* function (Def. 3.3.32) measures distances between two time points in terms of granule length. A distance between t_0 and t_1 may, for example, be 2.5 working days. We now generalise the notion of distance in terms of granules to the concept of *Durations*.

Definition 3.3.37 (Duration) *A Duration in PartLib is a sequence $D = (d_0\ G_0, \ldots, d_{n-1}\ G_{n-1})$ of pairs consisting of a real number d_i and a granularity G_i.*
The inverted duration *is* $-D \overset{\text{def}}{=} (-d_{n-1}\ G_{n-1}, \ldots, -d_0\ G_0)$ ■

1 month plus 1.5 days, for example, is represented as the sequence $(1\ \text{Month}, 1.5\ \text{Day})$ where Month and Day are granularities. 1 working day plus 1.5 hours, for example, could be represented in the same way $(1\ \text{WD}, 1.5\ \text{Hour})$.

A duration can also be used to specify a time shift. “Let us meet again in 3 months and 2 days”, for example, specifies a time shift in terms of a duration (3 month, 2 day). The corresponding shift function iterates through the duration and calls the shiftG function (Def. 3.3.36) for each granularity.

Definition 3.3.38 (Time Shift Function for Durations) *Let $D = (d_0\ G_0, \ldots, d_n\ G_{n-1})$ be a duration (Def. 3.3.37). We define a time shiftD function:*
$D.shiftD(time) \overset{\text{def}}{=} t_{n-1}$ *where*

1. $t_{-1} \overset{\text{def}}{=} time$
2. $t_i \overset{\text{def}}{=} G_i.shiftG(t_{i-1}, d_{i-1}, -1)$ *for* $0 \leq i < n$ *(Def. 3.3.36).* ■

3.3.7 Partitioning Types

The class **Partitioning** in PartLib is only the abstract interface for partitionings. Concrete partitionings are defined by sub-classing this class and providing the information for mapping partitions to partition coordinates and back. These are in particular the functions $PaCo(t)$ and $sPa(i)$. For the sake of efficiency some of the other functions of Def. 3.3.11 may be redefined as well.

3.3.8 Global and Local Reference Time, Leap Seconds

The global reference time GRT corresponds directly to UTC time. With the introduction of a *local reference time* LRT for each partitioning it is possible to deal with leap seconds. The purpose is that the algorithms for the different partitions can just use the local reference time, and do not need to deal with leap seconds. The leap second correction can therefore be done at a very deep level in the PartLib kernel.

A correction function for leap seconds is defined first. The function $lsG(t)$ defined below ('G' for 'global') computes the accumulated leap seconds until the global reference time point t. The function $lsL(t)$ ('L' for 'local') also computes the accumulated leap seconds, but until the 'local' reference time point t. $lsG(t)$ is used for the transition from GRT to LRT, whereas $lsL(t)$ is used for the other direction. Unfortunately, it is computationally difficult to derive one version from the other. It is much more efficient when both functions are generated from a table of leap second corrections. (see http://www.ptb.de/de/org/4/43/432/ssec.htm).

Definition 3.3.39 (Correction Function for Leap Seconds) *If t is a time point in the global reference time then $lsG(t)$ computes the accumulated number of leap seconds until t.*

If t is a time point in a reference time where the leap second corrections have already been done then $lsL(t)$ computes the accumulated number of leap seconds until t. ■

Example 3.3.40 (Correction Function for Leap Seconds) *The first 10 leap seconds were introduced for the last minute in 1971. The reference time for the regular end of this minute is 63072000. Therefore $lsG(t) = 0$ for all $t \leq 63072000$ and $lsG(63072000 + n) = n$ for $0 \leq n \leq 10$.*

$lsG(t)$ remains constant with value 10 from $t = 63072010$ until $t = 94694410$. $lsG(94694411) = 11$, because another leap second was introduced in the last minute of 1972.

$lsL(t) = 0$ for all $t \leq 63072000$ as well, but $lsL(63072001) = 10$. $lsL(94694400) = 10$ and $lsL(94694401) = 11$ etc.

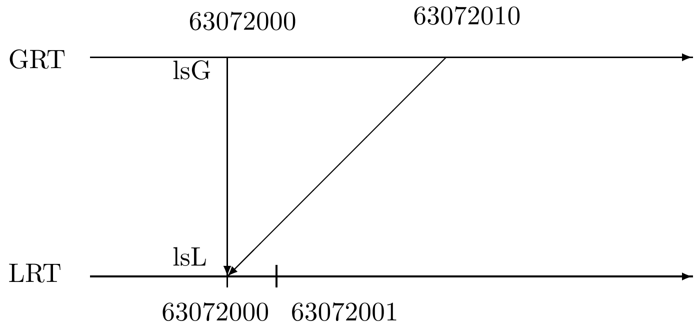


Figure 3.1: Global and Local Reference time around 31/12/1971

■

Definition 3.3.41 (Transition between GRT and LRT) *Given the correction functions lsG and lsL for leap seconds (Def. 3.3.39) we define*

$$LRT(t) \stackrel{\text{def}}{=} t - lsG(t).$$

for a global reference time t. $LRT(t)$ computes the local reference time from the global reference time.

The function

$$GRT(t) \stackrel{\text{def}}{=} t + lsL(t)$$

computes the global reference time from the local reference time. ■

The leap second corrections are automatically loaded at start time by the CTTN-system

3.3.8.1 Algorithmic Partitionings

These have been introduced in PartLib because there are partitionings whose details are so complex that a sophisticated algorithm is necessary to for computing $PaCo(t)$ and $sPa(i)$. Easter time is a prominent example. The algorithm which is currently in use for computing Easter time has first been presented by Karl Friedrich Gauss. It approximates the moon cycle to a certain degree. A pragmatic way to encode Easter time in a system like PartLib is to hard code the Gauss algorithm in a full fledged programming language like C++ and to compile it together with the whole PartLib–system. This is actually what happens when Algorithmic Partitionings are used[2].

It turns out that partitionings which seem so simple that they can be specified symbolically very easily need in fact also be encoded as Algorithmic Partitionings. Not all partitions of the day partitioning, for example, are exactly 24 hours long. When daylight savings time is used, one day in spring is only 23 hours long and one day in autumn is 25 hours long. The rules which control the switch from standard time to daylight savings time vary from country to country and they have been changed several times during the last 100 years. Therefore PartLib contains a special module which modifies the day, week and month partitionings such that daylight savings time is properly taken into account.

Definition 3.3.42 (Specification of Algorithmic Partitionings)
Algorithmic partitionings are specified in PartLib by the components (avl, po, cf) *where*

1. *avl is the average length of a partition, given in the finest time unit;*
2. *po is an offset for the partition with coordinate 0, also given in the finest time unit,*
3. $cf(i)$ *is a correction function.* ■

The correction function $cf(i)$ computes for a partition with coordinate i the difference between the reference time of the beginning of the partition with coordinate i, and the estimated beginning $i \cdot avl$.

Definition 3.3.43 (The Function $sPa_P(i)$ for Algorithmic Partitionings)
An algorithmic partitioning specification (avl, po, cf) *(Def. 3.3.42) generates the Coordinate Based Partitioning Representations* $(\mathbb{I}, sPa)$ *(Def. 3.3.6) as follows:*
1. The partition coordinates are the ordinary integers $\mathbb{I}$
2. The 'start of partition' function is:

$$sPa_P(i) \stackrel{\text{def}}{=} GRT(i \cdot avl + cf(i) + po).$$

■

Partitionings whose partitions have constant length in the local reference time only need a correction function that returns the constant 0. This is the case for seconds, minutes, and hours. It is no longer the case for days if daylight saving time regulations are taken into account.

[2]Algorithmic Partitionings could be implemented in a future version of PartLib as plugins.

Example 3.3.44 (Basic Time Units for the Gregorian Calendar)
The specification of the basic time units as algorithmic partitionings for the Gregorian Calendar are:

second: *average length: 1, offset: 0, correction function:* $\lambda(i)0$.

minute: *average length: 60, offset: 0, correction function:* $\lambda(i)0$.

hour: *average length: 3600, offset: 0, correction function:* $\lambda(i)0$.

day: *average length: 86400, offset: 0, correction function:* $-3600 \cdot h$ *if the day* i *is during the daylight saving time period, 0 otherwise.*
The number h *is usually 1 (for 1 hour). Exceptions are, for example, the year 1947 in Germany, where in the night of 1947/5/11 the clock was set forward a second time by 1 hour such that the offset against standard time was 2 hours.*

week: *average length: 604800, offset -259200, correction function: again, this function has to return an offset of* $-3600 \cdot h$ *for the weeks during the daylight saving time periods.*

month: *average length: 2592000 (30 days), offset 0, correction function: this function has to deal with the different length of the months and the daylight saving time regulations.*

year: *average length: 31536000 (365 days), offset 0, correction function: this function has to deal with leap years only. The effects of daylight saving time regulations are averaged out over the year.* ■

The 'partition coordinate' function $PaCo(t)$ maps a reference time point t to the coordinate of the partition containing t. For algorithmic partitionings this function is more complicated than $sPa_P(i)$ because it needs to use the correction function $cf(i)$, which takes a coordinate as input, and this is the coordinate which is yet to be computed. Therefore the basic idea for the algorithm is to use a fixed point iteration which calls $sPa_P(i)$ for guessed coordinates until the resulting time point matches the given time point. The algorithm is described rather informally, but the key steps should become clear.

Definition 3.3.45 (The Function $PaCo$ **for Algorithmic Partitionings)** *Let* t *be a local reference time point for the given partitioning* $P = (avl, po, cf)$. *The algorithm for PaCo (see Def. 3.3.11) starts with a first guess* $i \overset{\text{def}}{=} t/avl$ *for the coordinate of the partition containing* t. *Since this guess is wrong in general, there is a first iteration which brings* i *closer to the correct solution:*

Starting with an initial value for i', *a fixed point iteration is performed until* i' *falls under a certain threshold*[3]*: Let* $r \overset{\text{def}}{=} sPa_P(i)$. *If* $r \geq t$ *let* $r' \overset{\text{def}}{=} sPa_P(i-1)$ *and compute* $i' \overset{\text{def}}{=} (r-t)/(r-r')$ *to get a better estimate* $i \overset{\text{def}}{=} i - i'$ *for the correct coordinate. If* $r < t$, i *is increased in a similar way*[4].

The second phase of the algorithm is simpler: the correct coordinate is searched by just decreasing or increasing i *by 1, until* $sPa(i) \leq t < sPa(i+1)$ *holds. The result of the function* $PaCo(t)$ *is then the coordinate* i *for which this condition holds.* ■

During the first phase, the algorithm performs big jumps to get very close to the correct solution. In the second phase it does the fine tuning by searching in the neighbourhood of the

[3]The threshold in the implementation is 3. The initial value for i' can be any number greater than the threshold. $i' = 10$ is fine.

[4]This version of the fixed point iteration is slightly simplified. It can happen that i' oscillates around the correct i. If this happens, the iteration is immediately stopped.

coordinate which was computed in the first phase. This phase guarantees that the result is correct. The algorithm converges in very few (usually < 10) steps to the correct solution even if the average length of the partitions is quite different to their individual length.

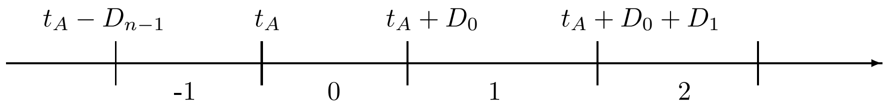
3.3.8.2 Duration Partitionings

Duration partitionings are specified by an anchor time and a sequence of Durations (Def. 3.3.37). For example, I could define 'my weekend' as a *duration partitioning* with anchor time 2004/7/23, 4 PM (Friday July, 23rd, 2004, 4 pm) and durations: ('8 hour + 2 day', '4 day + 16 hour'). The first interval would be labelled 'weekend', and the second interval would be labelled 'gap'.

Definition 3.3.46 (Specification of a Duration Partitioning)
A duration partitioning is specified by the tuple $(t_A, (D_0 \dots D_{n-1}))$ where

1. *t_A is the anchor time point (in the global reference time);*
2. *$D_0 \dots D_{n-1}$ is a list of durations (Def. 3.3.37);* ■

The coordinates for a duration partitioning are such that the first partition after the anchor time point has coordinate 0. The next picture illustrates the situation.



The durations in the specification of a duration partitioning can be very irregular. Therefore there is in general not much of a chance to realize a 'start of partition' function $sPa_P(i)$ other than by just looping i times over $D_0 \dots D_{n-1}$.

Definition 3.3.47 (The Function sPa for Duration Partitionings)
The specification $(t_A, (D_0 \dots D_{n-1}))$ (Def. 3.3.46) of a duration partitioning generates the following Coordinate Based Partitioning Representation $(IS_{\mathbb{I}}, sPa)$ (Def. 3.3.3):

1. *The partition coordinates are the ordinary integers $\mathbb{I}$*
2. *The'start of partition' function is $sPa(i) \overset{\text{def}}{=} t_i$ where t_i is determined by shifting the anchor time point t_A i times:*

   Let $t_0 \overset{\text{def}}{=} t_A$.

   if($i \geq 0$): for $j = 1, \dots, i$: $t_j \overset{\text{def}}{=} D_{(j-1)\%n}.shiftD(t_{j-1})$. (Def. 3.3.38)

   if($i < 0$): for $j = 1, \dots, -i$: $t_j \overset{\text{def}}{=} -D_{(n-j)\%n}.shiftD(t_{j-1})$ (Def. 3.3.38)

■

3.3.8.3 Date Partitionings

Some partitionings are determined by just giving a sequence of dates. An example could be the dates of the Time symposium series:

1994/5/4 Time94 1994/5/5, 1995/4/26 Time95 1995/4/27, ..., 2007/6/16 Time08 2005/6/18

There is a very easy way to deal with date partitionings: the dates are turned into partition boundaries, and the partitioning is represented explicitly (see Def. 3.3.72 below).

It is, however, also not very difficult to turn a sequence of dates into a sequence of durations. The distance between two subsequent dates is measured in terms of the partitionings which are used for expressing the dates. The result is turned into a Duration. Therefore Date Partitionings are just convenient ways to express Duration Partitionings. This way, however the originally finitely many partitions between the first and last date are extrapolated to the infinity. This may in fact be a desired effect, not in the case of the Time symposium series, but, for example, for defining the seasons approximatively by giving the beginning dates in one particular year: 2000/3/21 spring 2000/6/21 summer 2000/9/23 autumn 2000/12/21 winter 2001/3/21. The derived durations are: 3 months for spring, 3 months + 2 days for summer, 3 months + -2 days for autumn and 1 year + -9 month for winter. Notice that these durations are correct even for leap years. Although the period between March 2000 and March 2001 does not include a leap day, when going backwards from 2000/3/21 by one winter season (i.e. -1 year + 9 month) we jump over the leap day in February 2000 and end up at the correct date 1999/12/21.

All partitionings in PartLib can, however, be restricted by setting corresponding boundary times. This can be used to restrict the "validity region" of Date Partitionings to the interval between the first and last date.

The specification of date partitions requires a formal definition of *dates* in *date formats.* A typical date format is year/month/day/hour/minute/second, and a typical date is 2008/5/3/10/20/30.

Definition 3.3.48 (Date Formats and Dates) *A* Date Format *is a list $DF = (G_0, \ldots, G_n)$ of granularities.*
A Date *for a date format DF is a list $d = (d_0, \ldots, d_k)$ of integers, $k \leq n$.* ■

Notice that in this context dates are relative time shifts. Therefore they all start with the shift number 0. The (normalised) date for the first of January 2008 is therefore 2008/0/0. This way, a date like 2008/20/40 makes also sense. It means to shift the beginning of the year 2008 first by 20 month, and then further by 40 days.

Definition 3.3.49 (Date to Time) *Given a normalised date d (Def. 3.3.48), the time associated with the date is*

$$Time(d, DF) \stackrel{\text{def}}{=} D_d.shiftD(0)$$

where D_d is the associated duration (Def. 3.3.48). ■

Definition 3.3.50 (Time to Date) *Given a date format $DF = (G_0, \ldots, G_{n-1})$ and a time t, we define*

$$Date(time, DF) \stackrel{\text{def}}{=} (d_0, \ldots, d_{n-1})$$

where for $t_{-1} = 0$ and for $0, \leq i < n$:

$d_i \stackrel{\text{def}}{=} \lfloor G_i.lengthG(t_{i-1}, time) \rfloor + \begin{cases} -1 & \text{if } time < 0 \wedge i = 0 \\ 0 & \text{otherwise} \end{cases}$ *(Def. 3.3.32)*

$t_i \stackrel{\text{def}}{=} G_i.partitioning.sPa(d_i)$ ■

Dates like 2008/40/20 can be very easily normalised by first turning it into a time point, and then back into a date.

Definition 3.3.51 (Normalising Dates) *A date d for a date format DF can be normalised as follows:*

$$normalise(d, DF) \stackrel{\text{def}}{=} Date(Time(d, DF), DF)$$

■

Definition 3.3.52 (Date Partitioning Specification)
A Date Partitioning Specification $DPS = (DF, d_0, \ldots, d_m)$ *consists of a date format DF and a number of dates d_i in this format.* ■

Definition 3.3.53 (Date Differences) *For two dates $a = (a_0, \ldots, a_k)$ and $b = (b_0, \ldots, b_k)$ in a common date format $DF = (G_0, \ldots G_{n-1})$ we define the* date difference

$$d_0 -_{DF} d_1 \stackrel{\text{def}}{=} ((a_0 - b_0)G_0, \ldots, (a_k - b_k)G_k)$$

■

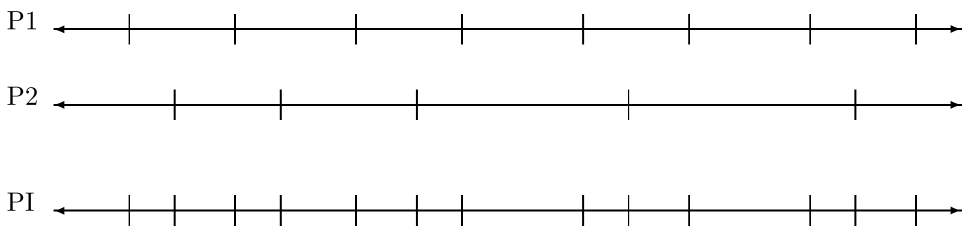
Definition 3.3.54 (Generated Duration Partitioning) *A Date Partitioning Specification $DPS = (DF, d_0, \ldots, d_m)$ (Def. 3.3.52) generates the following Duration Partitioning specification (Def. 3.3.46):*

1. $t_A \stackrel{\text{def}}{=} Time(d_0, DF_0)$ *(Def. 1.6.2)*
2. $D_i \stackrel{\text{def}}{=} d_{i+1} -_{DF} d_i$ *for* $0 \leq i < m$ *(Def. 3.3.53)*

■

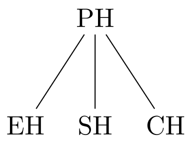
3.3.8.4 Intersection Partitionings

Two partitionings P1 and P2 can be joined into a new partitioning by just transferring the partition boundaries of P1 and P2 to the new partitioning PI, the "intersection Partitioning".



This idea alone may not seem very useful, but if combined with a suitable labelling scheme one can model interesting situations.

Suppose there is a partitioning PL for "lecture times" at the university, say every Monday from 10 AM - 12 AM labelled LM (Lecture Monday) and another one every Wednesday from 12 AM - 2 PM, labelled LW (Lecture Wednesday). The intervals between the lectures are not labelled. If we want to implement "lecture times outside public holidays", we need a partitioning for the public holidays. The holiday periods may be labelled EA (Easter Holidays), SH (Summer Holidays) and CH (Christmas Holidays). These labels may be part of a label hierarchy:

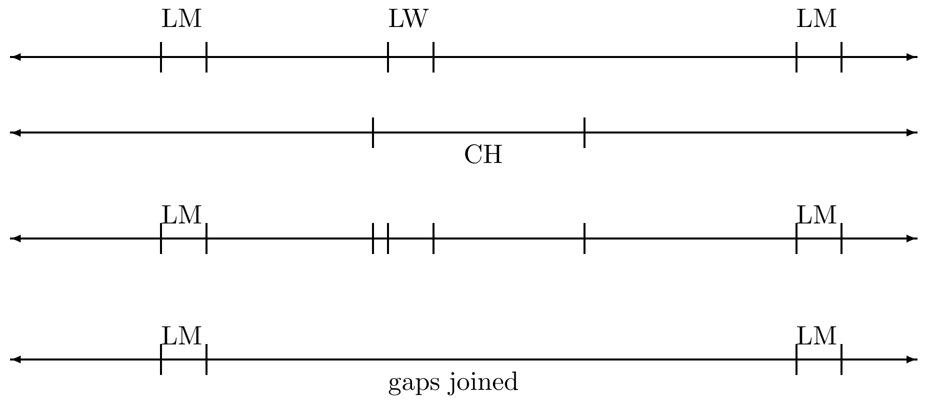


The two partitionings can now be intersected and the labels for the intersected partitioning can be computed according to the rules

$$\begin{aligned}
\text{LM} \times \text{gap} &\mapsto \text{LM}\\
\text{LM} \times \text{PH} &\mapsto \text{gap}\\
\text{LW} \times \text{gap} &\mapsto \text{LW}\\
\text{LW} \times \text{PH} &\mapsto \text{gap}
\end{aligned}$$

where 'gap' means, no label at all.

If all neighbouring gap partitions in the intersected partitioning are joined into one single gap partition one ends up with a partitioning which is like the 'lecture partitioning' PL, but the lectures during public holidays are removed.



PartLib offers the possibility to define "intersection partitionings" exactly in this way. The input consists of the two labelled component partitionings, a label hierarchy and a set of *mapping rules*. The two partitionings are intersected by transferring the partition boundaries to the new partitioning. The label of a given new partition is computed by applying the mapping rules to the two component partitions whose intersection forms the given new partition. Optionally, neighbouring gap partitions can be joined into one single gap partition.

Definition 3.3.55 (Label Mappings) *A label mapping $LM = (LH, (l_{0,1} \times l_{1,1} \mapsto l_{0,2}), \ldots))$ consists of a label hierarchy LH and an ordered list of* mapping rules $l_{0,1} \times l_{1,1} \mapsto l_{0,2}$.
The l_{ij} may be the empty label as well. ■

Definition 3.3.56 (Apply Function for Label Mappings)
For a label mapping $LM = (LH, (l_{0,1} \times l_{1,1} \mapsto l_{0,2}), \ldots))$ and two labels l_0 and l_1 we define:

$$LM.apply(l_0, l_1) \overset{\text{def}}{=} \begin{cases} l_{i,2} & \textit{if } i \textit{ is defined} \\ \emptyset & \textit{otherwise} \end{cases}$$

where $i = min\{i \mid l_0 \leq_{LH} l_{i,0} \wedge l_1 \leq_{LH} l_{i,1}\}$ ■

Definition 3.3.57 (Intersection Partition Specification) *An* Intersection Partitioning Specification *is a tuple* $(P_0, P_1, LM, joinGaps)$ *where the* P_i *are partitionings,* LM *is a label mapping (Def. 3.3.55) and* $joinGaps$ *is a Boolean flag.* ■

Definition 3.3.58 (Generated Intersection Partitioning) *An intersection partitioning specification* $(P_0, P_1, LM, joinGaps)$ *generates the following labelled partitioning* $P = (IS_P, sPa)$ *(Def. 3.3.4) where*

1. $sPa((P, lower, upper, l)) \stackrel{\text{def}}{=} lower$ *and*
2. $P.PaCo(time)$ *is defined in Def. 3.3.59 below.* ■

Definition 3.3.59 (Partition Coordinate for Intersection Partitionings)
Let P *be an intersection partitioning generated by* $(P_0, P_1, LM, joinGaps)$ *(Def. 3.3.58). We define*

$$P.PaCo(time) \stackrel{\text{def}}{=} (P, lower, upper, l)$$

where

1. $lower' \stackrel{\text{def}}{=} max(P_0.sPa(time), P_1.sPa(time))$
   $upper' \stackrel{\text{def}}{=} min(P_0.ePa(time), P_1.ePa(time))$
2. $l_0 \stackrel{\text{def}}{=} P_0.Label(P_0.PaCo(time))$,
   $l_1 \stackrel{\text{def}}{=} P_1.Label(P_1.PaCo(time))$ *and*
   $l \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } LM = \emptyset \\ LM.apply(l_0, l_1) & \text{otherwise} \end{cases}$
3. $lower \stackrel{\text{def}}{=} \begin{cases} lower' \quad \text{if } \neg joinGaps \vee l \neq \emptyset \\ max\{t < lower' \mid LM.apply(k_0, k_1) \neq \emptyset\} + 1 \;\; \text{otherwise} \\ \qquad \text{where } k_0 \stackrel{\text{def}}{=} P_0.Label(P_0.PaCo(t)) \text{ and } k_1 \stackrel{\text{def}}{=} P_1.Label(P_1.PaCo(t)) \end{cases}$

   $upper \stackrel{\text{def}}{=} \begin{cases} upper' \quad \text{if } \neg joinGaps \vee l \neq \emptyset \\ min\{t \geq upper' \mid LM.apply(k_0, k_1) \neq \emptyset\} \;\; \text{otherwise} \\ \qquad \text{where } k_0 \stackrel{\text{def}}{=} P_0.Label(P_0.PaCo(t)) \text{ and } k_1 \stackrel{\text{def}}{=} P_1.Label(P_1.PaCo(t)) \end{cases}$

■

3.3.8.5 Shifted Partitionings

Suppose a bus timetable for a particular bus station has been defined by means of a Tree Partitioning (Sect. 3.3.8.6). The bus time table for the next bus station may be such that all time are just shifted by, say, 5 minutes. It would be very inconvenient to force a user to specify this new timetable in the same complicated way as the first one. Moreover, if the original bus timetable is changed, the timetables for all other stations have to be changed as well. A much easier and safer way would be to define the other timetables by taking the first one and just shifting it by a certain duration.

To this end a partitioning type *Shifted Partitioning* has been introduced. It is specified by a given partitioning and a duration. All partitions of the new partitioning, together with their labels, are generated by shifting the original partitions by the given duration.

Definition 3.3.60 (Shifted Partitioning Specification) *A specification* (P, D) *for a* shifted partitioning *consists of a partitioning* P *and a duration* D. ■

Definition 3.3.61 (Generated Shifted Partitionings) *A shifted partitioning specification* (P, D) *generates the following Coordinate Based Partitioning Representation* $(P.CO, sPa)$ *with*

$$sPa(i) \stackrel{\text{def}}{=} D.shift(P.sPa(i)).$$

Other important functions are:

$$\begin{array}{lll} PaCo(time) & \stackrel{\text{def}}{=} & P.PaCo(-D.shift(time))) \\ sPa(time) & \stackrel{\text{def}}{=} & sPa(PaCo(time)) \\ sPa(i) & \stackrel{\text{def}}{=} & D.shift(P.sPa(i)) \\ ePa(time) & \stackrel{\text{def}}{=} & D.shift(P.ePa(-D.shift(time))) \\ Label(i) & \stackrel{\text{def}}{=} & P.Label(i) \end{array}$$

■

3.3.8.6 Tree Partitionings

Many practical partitionings of the time axis can be specified as a kind of *iterator*. For example, a specification of a bus timetable could be:

"In the year 2008, every week, every work day (0–4), there is a bus at 5:20 – 5.21 (2 minutes stay at the bus stop), at 5:40 – 5:41, at 6:20 – 6:21, at 6:40 - 6:41 until 20:40 – 20:41, and at the weekends (days 5,6) there is a bus every hour from 8 until 16 hours".

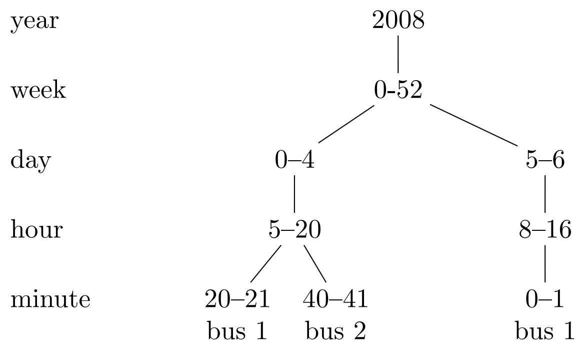
In order to express this specification in a formal system, we need the following ingredients:

1. A date format (Def. 3.3.48) that specifies the meaning of "year , "week" etc.

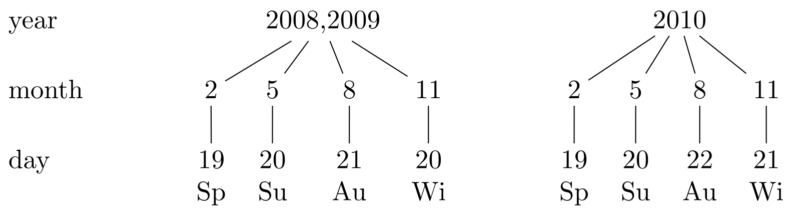
2. For each level of the date format there must be a kind of iterator which expresses a sequence of coordinates or coordinate offsets. 'every week', for example must be represented by an iterator which generates all the week numbers within a year. If 'every working day' means, days 0-4 within a week, then "0–4" is a specification of an iterator which represents the days 0,1,2,3,4 within a week. In this case one could also want to use instead of numbers the corresponding labels "Monday – Friday" with the same meaning. PartLib allows one to use both versions. "0–4" is a very simple iterator. In more complex examples, one might use iterators like "five times every second day, starting with day 3", which is almost like a for-loop in programming languages. PartLib provides iterators of this kind, both for numbers and for labels.

3. The specification language must be able to distinguish different cases, e.g. "at working days" and "at the weekend". Therefore the specification may have a tree structure, hence the name *tree partitionings*.

The tree specification of the bus timetable example could be as follows:

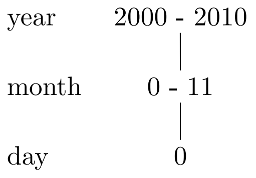


A characteristic feature of this example is that each leaf node specifies an interval with a particular meaning (bus is at bus stop). The intervals between the ‘bus stop intervals’ are just gaps. There are, however, other examples where each leaf node specifies the start of a partition which lasts up to the start time specified by the next leaf node. An example, where this is the case are the seasons. For the years 2008-2010 they could be specified by the following trees[5]:



The different meaning of the leaf nodes are controlled by a Boolean parameter “continuous”. *continuous* $= true$ means that the partition specified by a particular leaf node lasts from the start time of this leaf node up to the start time of the next leaf node. If the leaf node has a label, this is the label for this partition. *continuous* $= false$ would be needed for the bus timetable example. It specifies that each leaf node determines a partition, and this is labelled by the leaf node’s label. The gaps between these partitions have no label.

The bus example and the seasons example above specify partitionings where the partitions get newly defined labels “bus 1”, “bus 2” “Sp” etc. In other examples, it may make sense to take the labels for the new partitions from the labels of the granularities in the date format. Consider an iterator which enumerates the first days of each month in some years:



If the day partitioning is labelled “Mo”, “Tu” etc., we may want to have day labels as labels of the new partitions. With a corresponding control parameter “granuleLabel” set to “true”, we get a partitioning which consists of a day partition for the first day in a month, followed by a large partition for the remaining days in the month. The day partition as label the corresponding day name.

PartLib’s tree partitioning specification mechanism consists of the three components:

[5]Months and days are counted from 0.

1. A date format DF (Def. 3.3.48);

2. The Boolean "continuous" parameter;

3. A list of Range Trees (RT), which are trees whose n'th level correspond to the n'th granularity in DF. Each node contains an iterator which iterates over relative granule positions or over labels. The leaf nodes of the trees define the length of the corresponding partitions. They can also be labelled. (In the bus timetable example the labels at the leaf nodes could be the bus numbers). The meaning of the leaf nodes is controlled by the "continuous" parameter (see above). The label associated with the interval specified by the leaf node is either given explicitly or taken from the corresponding granularity in the date format.

Definition 3.3.62 (Tree Partitioning Specification (TPS))
A Tree Partitioning Specification *is a triple* $TPS \overset{\text{def}}{=} (DF, subtrees, continuous)$ *where*

1. DF is a date format (Def. 3.3.48),
2. subtrees $= (RT_0, \ldots, RT_n)$ *is a list of Range Trees (see below) and*
3. continuous is a Boolean flag.

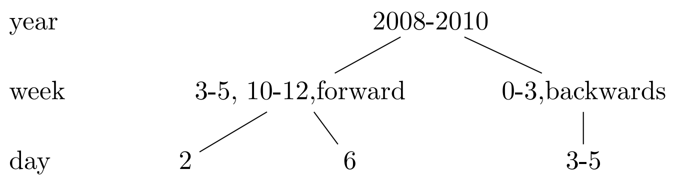
A Range Tree *is tuple* $RT \overset{\text{def}}{=} (RTNode, subtrees)$ *where*

1. RTNode is a Range Tree Node *(see below) and*
2. subtrees $= (RT_0, \ldots, RT_{k-1})$ *is a, possibly empty, list of Range Trees.*

A Range Tree Node $RTNode = (NPositions, label, granuleLabel)$ *consists of a function* $NPositions : \texttt{Time}^2 \times Granularity^2 \mapsto FiniteSet(\mathbb{I})$ *a, possibly empty, label, and a Boolean parameter granuleLabel. "label" and "granuleLabel" are relevant only for leaf nodes.*

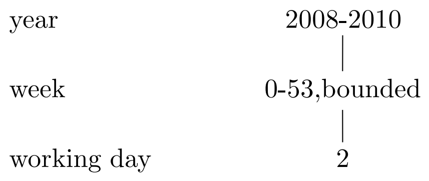
The depths of the Range Trees must be $\leq |DF|$. ■

The *NPositions* function for Range Tree Nodes allows one to specify relative granule positions in a convenient way and to normalise the positions automatically. To this end there are different Range Tree Node types, for which different types of *NPositions* functions are automatically generated. One of them are the Number Range Nodes (Def. 3.3.77). All nodes in the Tree Partitioning Specification below are of this type.



The *NPositions* function returns for the year node the numbers {2008, 2009, 2010}, For the left week node it returns {3, 4, 5, 10, 11, 12}. The result for the second week node, where weeks are counted backwards, depends on the total number of weeks in the corresponding year. $NPositions(t_0, t_1, G_{week}, G_{year})$ is called where $[t_1, t_2[$ is one of the years 2008, 2009 and 2010. The length of the year determines the number of weeks in this year. If there are 52 weeks in the year then the result of *NPositions* is {48, 49, 50, 51}. If there are 53 weeks in the year then {49, 50, 51, 52} is returned.

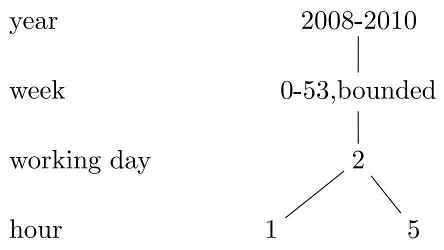
The meaning of the Tree Partitioning Specification is quite obvious when the date format consists of granularities like year/month/day etc. Date formats can, however, consists of granularities with internal and external gaps. To illustrate this, consider the specification “in the years 2008-2010, in every week, in every second working day”. A Tree Partitioning Specification for this could be



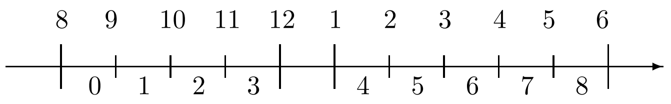
(The parameter *bounded* restricts the week iteration 0-53 to the actual number of weeks within a given year.)

Suppose, a working day lasts from 8 AM until 6 PM with a one hour lunch break at noon. The question is now, whether the lunch break should be part of the newly defined partitions or not. The decision in PartLib is that internal gaps are ignored for granularities at leaf node positions in a Tree Partitioning Specification. That means in this case that the whole working day from 8 AM until 6 PM becomes a single partition.

Internal gaps, however, are not ignored when the granularities occur at higher positions in the Tree Partitioning Specification. To illustrate this, consider a slightly extended specification: “in the years 2008-2010, in every week, in every second working day, in every 1st and 5th hour”. The corresponding Tree Partitioning Specification would be



The question is now, how the hours are counted inside a working day. Does the lunch break increase the hour counter or not? The decision in PartLib is that internal gaps are not counted. That means in this example, hour 5 is between 2 PM and 3 PM, as indicated in the figure below.



Therefore “hour 1 and hour 5” within a working day are between 9 AM and 10 AM and between 2 PM and 3 PM. Technically this counting mechanism with internal gaps is realised in the function $G.nthGranule(t, n, G')$ (Def. 3.3.30) where the internal gaps of the granularity G' influence the counting of G-granules.

The label associated with a leaf node of a TPS is either given explicitly or taken from the corresponding granularity of the date format. Therefore we define a function *Label* for an RTNode which returns the associated label.

Definition 3.3.63 (Label for an RTNode)
For an RTNode $RTN = (NPositions, label, granuleLabel)$, a time point time and a granularity G we define the function

$$RTN.Label(time, G) \overset{\text{def}}{=} \begin{cases} label & \text{if } granuleLabel = false \\ P.Label(P.PaCo(time)) & \text{otherwise} \end{cases}$$

where $P \overset{\text{def}}{=} G.partitioning$. ■

The auxiliary function *NInterval* below computes for an RTNode the interval boundaries specified by this node. For example, if at the month level, the RTNode specifies month 3-5 (in, say, year 2008) then *NInterval* computes the interval from the start of month 3 up to the end of month 5 in this year. The function is used in Def. 3.3.66.

The second function which is defined below, called *NPartitioning*, computes for a leaf node a representation of a corresponding labelled partitioning as a list $((interval_0, label_0), \ldots)$. The partitioning depends on the parameter *continuous*. If $continuous = true$ then the partitioning consists of just two partitions $]-\infty, a[, [a, +\infty]$ where a is the start time of the first granule specified by the leaf node. If $continuous = false$ then the partitioning consists of three partitions $]-\infty, a[$, $[a, b[$, $[b, +\infty]$ where $[a, b[$ is the convex hull of the granules specified by the leaf node.

Definition 3.3.64 (*NInterval* and *NPartitioning*)
Given an RTNode $RTN = (NPositions, label, granuleLabel)$, we define for a time interval $[t_0, t_1[$ and two granularities G, G' a function

$$RTN.NInterval(t_0, t_1, G, G') \overset{\text{def}}{=} \begin{cases} \emptyset & \text{if } RTN.NPositions(t_0, t_1, G, G') = \emptyset \\ [a, b[& \text{otherwise} \end{cases}$$

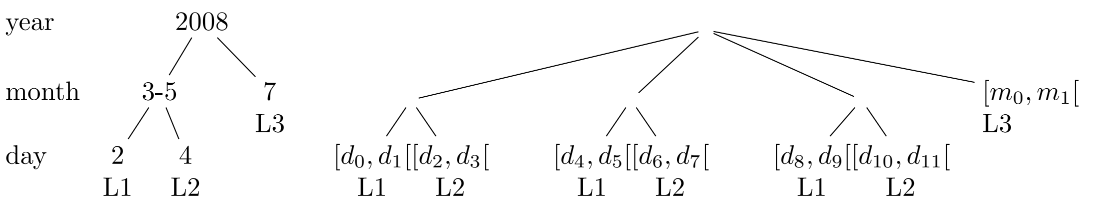
where
1. $i \overset{\text{def}}{=} min(NPositions(t_0, t_1, G, G'))$ *and* $j \overset{\text{def}}{=} max(NPositions(t_0, t_1, G, G'))$,
2. $[a, c[\overset{\text{def}}{=} G.nthGranuleI(t_0, i, true, G')$ *and* *(Def. 3.3.30)*
3. $[d, b[\overset{\text{def}}{=} G.nthGranuleI(t_0, j, true, G')$.

Furthermore we define a function NPartitioning which generates for a leaf node RT a corresponding partitioning.

$$RTN.NPartitioning(t_0, t_1, G, G', continuous) \overset{\text{def}}{=}$$
$$\begin{cases} ((]-\infty, +\infty[, \emptyset) & \text{if } RTN.NPositions(t_0, t_1, G, G') = \emptyset \\ ((]-\infty, a[, \emptyset), ([a, +\infty[, RTN.Label(a, G))) & \text{if } continuous = true \quad \text{(Def. 3.3.63)} \\ ((]-\infty, a[, \emptyset), ([a, b[, RTN.Label(a, G))), ([b, +\infty[, \emptyset)) & \text{otherwise} \end{cases}$$

where $[a, b[\overset{\text{def}}{=} RTN.NInterval(t_0, t_1, G, G')$ ■

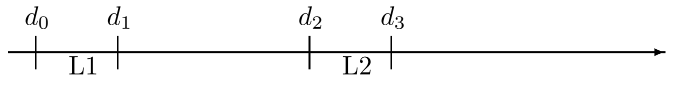
The Semantics of Tree Partitioning Specifications. Tree Partitioning Specifications (TPS, Def. 3.3.62) are supposed to specify labelled partitionings. Therefore there must be a mapping from TPS to partitionings. This mapping can be defined in two steps. The first step transforms a TPS into an *Expanded Tree Partitioning Specification* (ETPS) where all nodes representing several granules are expanded to one node per granule, and the leaf nodes contain concrete time intervals. The TPS at the left side below, for example, is expanded to the ETPS at the right hand side:



where $[m_0, m_1[$ is month 7 in the year 2008, $[d_0, d_1[$ = day 2 in month 3 of the year 2008, $[d_2, d_3[$ = day 4 in month 3 of the year 2008, $[d_4, d_5[$ = day 2 in month 4 of the year 2008, etc.

Notice that the ETPS does not distinguish between continuous TSP and non-continuous TPS.

The second transformation, from Expanded TPS to partitionings, associates a partitioning to each leaf node of the ETPS, and intersects the partitionings associated to sub-nodes to get corresponding partitionings for higher level nodes. The partitioning associated with leaf node $([d_0, d_1[$,L1) in the above example is $(]-\infty, d_0[, \emptyset)$, $([d_0, d_1[$,L1), $([d_1, +\infty[, \emptyset))$ ($continuous = false$ assumed). The partitioning associated with leaf node $([d_2, d_3[$,L2) in the above example is $(]-\infty, d_2[, \emptyset)$, $([d_2, d_3[$,L2), $([d_3, +\infty[, \emptyset))$. The intersected partitioning is then $(]-\infty, d_0[, \emptyset)$, $([d_0, d_1[$,L1), $([d_1, d_2[, \emptyset)$, $([d_2, d_3[$,L2), $([d_3, +\infty[, \emptyset))$.



By intersecting the partitionings associated with sub-nodes of a given node in an ETPS, one finally gets the partitioning associated with the ETPS itself.

Definition 3.3.65 (Expanded Tree Partitioning Specification (ETPS))
An Expanded Tree Partitioning Specification (ETPS) is a tuple $((ERT_0, \ldots, ERT_n), continuous)$ *where the* ERT_i *are Expanded Range Trees, either of the form* $ERT_i \stackrel{\text{def}}{=} subnodes$ *with a list of Expanded Range Trees as sub-nodes, or as leaf nodes of the form* $([a, b[, label)$.
'continuous' is a Boolean parameter. ■

The function $\mathcal{E}$ in Def. 3.3.66 below maps a TPS to an Expanded TPS.

Definition 3.3.66 (From TPS to ETPS) *We define a function* $\mathcal{E}$ *which maps a Tree Partitioning Specification* $(DF, subtrees, continuous)$ *(Def. 3.3.62) to an Expanded Tree partitioning Specification (Def. 3.3.65):*

$$\mathcal{E}((DF, subtrees, continuous)) \stackrel{\text{def}}{=} (\mathcal{E}(subtrees, 0, +\infty, DF, 0), continuous)$$

where
$\mathcal{E}((RT_0, \ldots, RT_n), t_0, t_1, DF, level) \stackrel{\text{def}}{=}$
$\{\mathcal{E}(RT_k.subtrees, a, b, DF, level + 1) \mid k \in \{0, \ldots, n\}$ *with*
$RT_k.subtrees \neq \emptyset, i \in RT_k.RTNode.Npositions(t_0, t_1, G, G')$,
$[a, b[\stackrel{\text{def}}{=} G.nthGranuleI(t_0, i, true, G')\} \cup$
$\{([a, b[, RT_k.RTNode.Label(a, G)) \mid RT_k.subtrees = \emptyset$,
$[a, b[= RT_k.RTNode.NInterval(t_0, t_1, G, G') \neq \emptyset, k \in \{0, \ldots, n\}\}$

and $G \overset{\text{def}}{=} DF[level]$ *and* $G' \overset{\text{def}}{=} \begin{cases} DF[level-1] & \textit{if level} > 0 \\ \emptyset & \textit{otherwise} \end{cases}$ ■

The Expanded Tree Partitioning Specification is still a finite structure, but the number of leaf nodes can grow exponentially with the depth of the tree, compared to the Tree Portioning Specification itself. A simple example shows this. The TPS

$$\begin{tikzcd}[row sep=small]
\text{year} & \text{2008} \arrow[d, dash] \\
\text{month} & \text{0-11} \arrow[d, dash] \\
\text{day} & \text{0-31} \arrow[d, dash] \\
\text{hour} & \text{0-23}
\end{tikzcd}$$

expands to $1 \cdot 12 \cdot 32 \cdot 24$ leaf nodes which is exponential in the depth of the tree.

Proposition 3.3.67 (Exponential Size of ETPS) *The number of leaf nodes of an Expanded Tree Partitioning Specification is in the order $\mathcal{O}((n*m)^{|}DF|)$ where n is the maximal number of relative granule positions the NPositions function generates, and m is the maximum number of sibling nodes in the original TPS.*

The proof is obvious. ■

Notice that an Expanded TPS may become empty. An example where this happens is the TPS

$$\begin{tikzcd}[row sep=small]
\text{year} & \text{2008} \arrow[d, dash] \\
\text{day} & \begin{matrix}\text{holiday} \\ \text{L}\end{matrix}
\end{tikzcd}$$

where the day node is a Label Range Node (Def. 3.3.79) below) and there is no day with label "holiday" at all. The corresponding partitioning consists of the entire time line as a single partition with empty label.

The second step, the mapping from ETPS to partitionings requires to intersect two labelled partitionings. In the corresponding definition (Def. 3.3.68 below) we exploit that the Tree Partitioning Specifications specify only finite partitionings with infinite (unlabelled) start partition and infinite (unlabelled) end partition, Thus, a labelled partitioning can be described as a finite sequence of tuples $(]-\infty, t_0[, \emptyset)$,$([t_0, t_1[, L_1)$,..., $([t_{n-1}, t_n[, L_n)$, $([t_n, +\infty[, \emptyset)$, where the L_i are, possibly empty, labels.

Definition 3.3.68 (Intersection of Finite Labelled Partitionings) *Given two finite labelled partitionings*
$P = (]-\infty, a_0[, \emptyset)$,$([a_0, a_1[, L_1)$,..., $([a_{n-1}, a_n[, L_n)$, $([a_n, +\infty[, \emptyset)$ *and*
$Q = (]-\infty, b_0[, \emptyset)$,$([b_0, b_1[, N_1)$,..., $([b_{m-1}, b_m[, N_m)$, $([b_m, +\infty[, \emptyset)$
we define the intersection
$P \cap Q \overset{\text{def}}{=} (]-\infty, c_0[, \emptyset)$,$([c_0, c_1[, M_1)$,..., $([c_{n+m+1}, c_{n+m+2}[, M_{n+m+2})$, $([c_{n+m+2}, +\infty[, \emptyset)$ *where*
1. $c_0, \ldots, c_{n+m+2} \overset{\text{def}}{=} \textit{sorted}(\{a_0, \ldots, a_n, b_0, \ldots, b_m\})$ *and*

2. the labels M_i are defined as follows:

Case 1 *$c_i = a_j = b_k$ for some $j \in \{0, \ldots, n\}$ and some $k \in \{0, \ldots, m\}$:*

$$M_i \overset{\text{def}}{=} \begin{cases} L_j & \text{if } N_k = \emptyset \vee (L_j \neq \emptyset \wedge a_{j+1} - a_j \leq b_{k+1} - b_k) \\ N_k & \text{otherwise} \end{cases}$$

Case 2 *$c_i = a_j$ and $b_k < c_i < b_{k+1}$ for some $k \in \{0, \ldots, m\}$:*

$$M_i \overset{\text{def}}{=} \begin{cases} L_j & \text{if } L_j \neq \emptyset \\ N_k & \text{otherwise} \end{cases}$$

Case 3 *$c_i = b_k$ and $a_j < c_i < a_{j+1}$ for some $j \in \{0, \ldots, n\}$:*

$$M_i \overset{\text{def}}{=} \begin{cases} N_k & \text{if } N_k \neq \emptyset \\ L_j & \text{otherwise.} \end{cases}$$

(Case 2 and 3 are symmetric.)

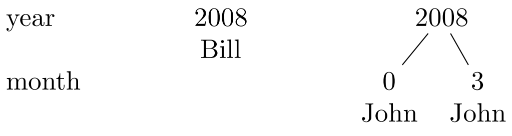
For a set $\{P_0, \ldots, P_{n-1}\}$ of finite labelled partitionings we define

$$\begin{aligned} \bigcap\{\} &\overset{\text{def}}{=} (]-\infty, +\infty[, \emptyset) \\ \bigcap\{P_0\} &\overset{\text{def}}{=} P_0 \\ \bigcap\{P_0, \ldots, P_{n-1}\} &\overset{\text{def}}{=} P_0 \cap \bigcap\{P_1, \ldots, P_{n-1}\} \end{aligned}$$

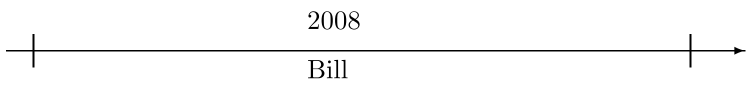
■

The rationale behind the rules for defining the labels for the intersected partitioning $P \cap Q$ can be illustrate with some examples.

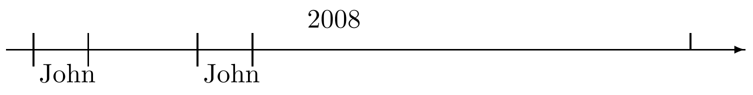
Consider the example: "In the year 2008, Bill is responsible, except for the months 0 and 3, where John is responsible (for something)" This could be represented as the TPS:



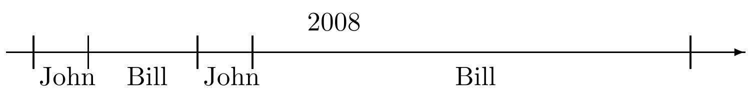
The left node yields the partitioning



The right node yields the partitioning



The intersection yields

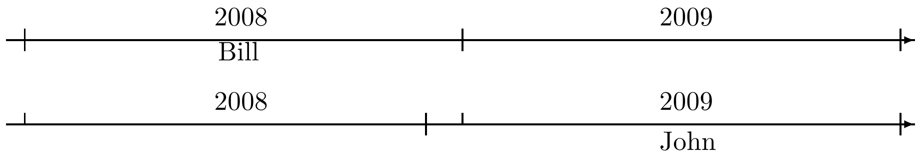


For month 0 in the year 2008, there was a choice between label Bill and label John. The heuristic here is: the label of the intersected partition is the one of the smallest (most specific) partition (Case 1 in Def. 3.3.68). The same heuristic yields Bill as label for month 3.

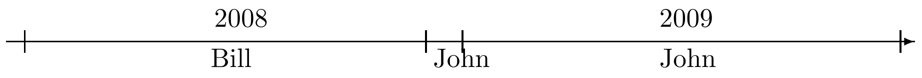
The next example illustrates another heuristic. "In the year 2008, Bill is responsible, except for December and the whole year 2009, where John is responsible". This can be represented as the TPS:

year	2008	2009
	Bill	|
month		-1-11
		John

The corresponding component partitionings are



with the intersection partitioning



In December there was a choice between Bill and John. The heuristic here is: If we go from past to the future and start a new partition (start of December) then the label of the newer partition (if there is one) wins (Case 2 and 3 in Def. 3.3.68).

As a side effect, John's partition is split into two intervals, the December 2008 and the whole year 2009. If this is not wanted one must avoid intersecting partition, and specify, for example:

year	2008	2009
	|	|
month	0-10	-1-11
	Bill	John

Remark 3.3.69 (Complexity of ∩) *The intersection of two (or more) finite labelled partitionings (Def. 3.3.68) needs to sort the lists of partition boundaries. Here we can exploit that the lists are already sorted. Therefore joining and sorting the lists can be done in linear time by a sweep line algorithm (a simultaneous sweep through the lists from left to right). The labels of the intersected partitions can also be computed in the same sweep. Therefore the complexity of the intersection procedure for finite labelled partitionings is linear in the number of partition boundaries.* ■

Notice that there is an element of randomness in the definition of ∩ (Def. 3.3.68) which can destroy the commutativity of the ∩-operation for partitionings. The following TPS, which is legal, but which makes not much sense, illustrates the phenomenon.

year	2008	2008
	L1	L2

It specifies two different labels for the same year 2008. The intersection operation for the corresponding partitionings has to choose between the labels L1 and L2. According to Def. 3.3.68 it chooses L1, just because it goes from left to right. This, however, is just by chance and could be different in different implementations. Therefore TPS like this one should be avoided.

We can now define a mapping $\mathcal{P}$ from ETPS to partitionings. It generates partitionings for the leaf nodes of an ETPS and works upwards to the roots by intersecting the partitionings associated to the sub-nodes. The difference between *continuous* = *true* and *continuous* = $false$

shows up now at the leaf nodes. A leaf node $([t_0,t_1[,L)$ yields a partitioning $(]-\infty,t_0[,\emptyset)$, $([t_0,t_1[,L)$, $([t_1,+\infty[,\emptyset)$ if $continuous = false$ and $(]-\infty,t_0[,\emptyset[)$, $([t_0,+\infty[,L)$ if $continuous = true$ (the upper bound t_1 is ignored).

Definition 3.3.70 (From ETPS to Partitionings)
Given a $ETPS = ((ERT_0,\ldots,ERT_n), continuous)$ (Def. 3.3.65) we define a mapping

$$\mathcal{P}(ETPS) \stackrel{\text{def}}{=} \bigcap\{\mathcal{P}(ERT_0),\ldots,\mathcal{P}(ERT_n)\}$$

where for the leaf nodes

$$\mathcal{P}([t_0,t_1[,L) \stackrel{\text{def}}{=} \begin{cases} (]-\infty,t_0[,\emptyset),([t_0,+\infty[,L) & \textit{if continuous} = true \\ (]-\infty,t_0[,\emptyset),([t_0,t_1[,L),([t_1,+\infty[,\emptyset) & \textit{otherwise} \end{cases}$$

and for the other nodes

$$\mathcal{P}((ERT_0,\ldots,ERT_n)) \stackrel{\text{def}}{=} \bigcap\{\mathcal{P}(ERT_0),\ldots,\mathcal{P}(ERT_n)\} \qquad \textit{(Def. 3.3.68)}$$ ■

The two functions $\mathcal{E}$ and $\mathcal{P}$ can be combined to generate A finite labelled partitioning for a TPS in a single operation. The recursive descent from the root nodes of a TPS down to the leaf nodes computes the ETPS, and the ascent back to the root nodes generates the partitionings.

Definition 3.3.71 (From TPS to Partitionings) *We define a function $\mathcal{P}a$ which generates from a Tree Partitioning Specification $(DF, subtrees, continuous)$ (Def. 3.3.62) a partitioning.*

$$\mathcal{P}a(DF, subtrees, continuous) \stackrel{\text{def}}{=} \mathcal{P}a(subtrees, 0, +\infty, DF, 0)$$

where

$$\begin{aligned}
&\mathcal{P}a((RT_0,\ldots,RT_n),t_0,t_1,DF,level) \stackrel{\text{def}}{=} \\
&\quad \bigcap\{\mathcal{P}a(RT_k.subtrees,a,b,DF,level+1) \mid k \in \{0,\ldots,n\} \textit{ with} \qquad \textit{(Def. 3.3.68)} \\
&\qquad RT_k.subtrees \neq \emptyset, i \in RT_k.RTNode.Npositions(t_0,t_1,G,G'), \\
&\qquad [a,b[\stackrel{\text{def}}{=} G.nthGranuleI(t_0,i,true,G')\} \cap \\
&\quad \bigcap\{RT_k.NPartitioning(t_0,t_1,G,G',continuous) \mid RT_k.subtrees = \emptyset\} \quad \textit{(Def. 3.3.64)}
\end{aligned}$$

and

$$G \stackrel{\text{def}}{=} DF[level] \textit{ and } G' \stackrel{\text{def}}{=} \begin{cases} DF[level-1] & \textit{if level} > 0 \\ \emptyset & \textit{otherwise} \end{cases}$$ ■

Although the number of partitions in a TPS generated partitioning (Def. 3.3.71) may be exponential (Prop. 3.3.67), it may be still small enough that it is possible to compute it explicitely for further use.

Definition 3.3.72 (Explicitely TPS-Generated Partitionings)
The $\mathcal{P}a$-function (Def. 3.3.71) can turn a Tree Partitioning Specification into a labelled partitioning

$$([-\infty,t_0[,\emptyset),([t_0,t_1[,L_0),\ldots,([t_{n-1},t_n[,L_{n-1}),([t_n,+\infty[,\emptyset)$$

which can be represented by two arrays:

$$\begin{array}{lcccccc}
Times & \stackrel{\text{def}}{=} & -\infty & t_0, & t_1, & \ldots, & t_n, & +\infty \\
Labels & \stackrel{\text{def}}{=} & \emptyset & L_0, & L_1, & \ldots, & L_n, & \emptyset
\end{array}$$

where

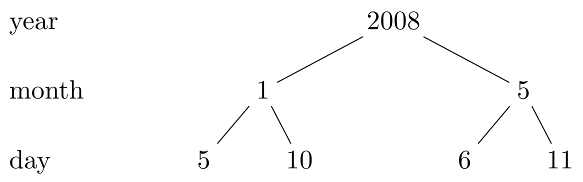
1. *the partition coordinates are the indices into the arrays (natural numbers with 0);*
2. *the sPa function needs to access the Times array:* $sPa(i) \stackrel{\text{def}}{=} Times[i]$*;*
3. *the Label function needs to access the Labels array:* $Label(i) \stackrel{\text{def}}{=} Labels[i]$*, and*
4. *the PaCo function needs to locate for a time point t the array element* $PaCo(t) \stackrel{\text{def}}{=} i$ *with* $Times[i] \leq t < Times[i+1]$*. This can be done with binary search in* $\mathcal{O}(log\ n)$ *time (because the Times array is sorted).*

■

Computationally, the explicit representation of the TPS-generated partitioning is most efficient. The *PaCo*-function works in $\mathcal{O}(log\ n)$ time, and all the other key functions work in constant time. Unfortunately, the generated partitioning may be exponential in the size of the TPS. Therefore, for large partitionings, it may be necessary to work at the original TPS directly.

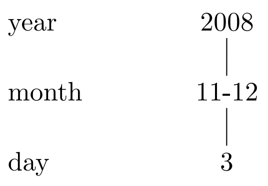
Tree algorithms, however, are only efficient if they can focus on a very few number of branches in the tree. Examining all branches requires in general exponential time. It turns out that a particular property, boundedness of nodes, in a TPS is crucial for the efficiency of the tree algorithms for TPS.

Bounded Nodes: The *PaCo* function has to compute for a time point t the partition containing this point. This is the key function for a number of other functions and should therefore be realised by an efficient algorithm. A tree structure as in a Tree Partitioning Specification (Def. 3.3.62) should be an ideal basis for an efficient *PaCo* algorithm. For simple specifications like the one below this is in fact the case.



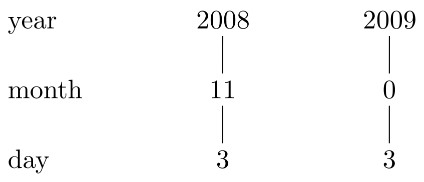
Suppose a time point t is in day 6 of month 5 in the year 2008. An efficient *PaCo* algorithm would start at the root node, 2008, and check whether t is in 2008. This is easy to check and yields *true* for t. One level down the algorithm checks whether t is in month 1 or in month 5. The check for month 5 yields true, and therefore the subtree below 1 can be discarded from the search. This is typical tree search procedure with logarithmic complexity.

Unfortunately, not all examples are so simple. A very natural Tree Partitioning Specification for "day 3 in December 2008 and January next year" would be:



(Month 11 is December and month 12 is January next year). One of the partitions is January 2009, which is *below* the year node 2008, but not *in* the year 2008. Therefore the simple tree search algorithm would not find this partition.

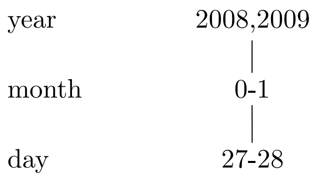
A simple solution would be to forbid such specifications. This,however, would force the user to split the tree:



A more user friendly solution would be to allow specifications where sub-nodes specify relative granule positions which are outside the time intervals specified by their super-nodes. A consequence is, however, that the tree search algorithm does not know which path to follow. It must check all paths.

A compromise between efficiency and user friendliness is to mark those nodes N whose subtrees stay within the bounds specified by N as *bounded*. The search algorithm can then ignore bounded nodes if a time point does not lie within the granules specified by the bounded nodes.

Unfortunately, the concept of boundedness of nodes in a TPS does not only depend on the node itself and its sub-nodes. It may also depend on the path to this node. The following example illustrates the phenomenon.

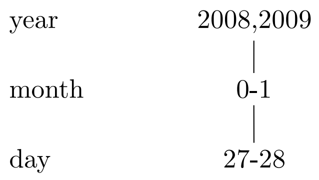


Since 2008 is a leap year, but 2009 is not, the month node is bounded only for 2008. Day 28 in February 2009 (which is the 29th of February) is outside this month. Therefore the month node is not (globally) bounded.

Boundedness of nodes can be ensured, or at least checked in different ways:

1. The *NPositions* function can automatically ensure boundedness. This is always the case for Label Range Nodes (Def. 3.3.79) and optionally for Number Range Nodes (Def. 3.3.77), Number Iterator Nodes (Def. 3.3.78) and Label Iterator Nodes (Def. 3.3.80).

2. It can be checked with a special algorithm. This algorithm marks every node in a TPS with an extra flag "bounded" and then runs a procedure similar to $\mathcal{E}$ (Def. 3.3.66) that computes for each expanded path to a node N the reference interval generated by N and compares it with the reference intervals generated by the sub-nodes of N. If there is a path to N such that the reference interval generated by N is smaller than the convex hull of the reference intervals of N"s sub-nodes, then N is marked "unbounded". The procedure can be illustrated with the above example:



The month node is marked *bounded* $= false$ after the following checks:
1. year 2008, month 0 yields *true* because January has 31 days.
2. year 2008, month 1 yields *true* because February has 29 days.
3. year 2009, month 0 yields *true* because January has 31 days.
4. year 2009, month 1 yields $false$ because February has 28 days, and the day node is outside this February.

The year node, however, gets "bounded = true" because both years have more than 2 months. If, however, the day node would be, for example, 'day 27-400' then the year node would also be unbounded.

Definition 3.3.73 (Boundedness Check) *We define a function $\mathcal{BC}$ (Boundedness Check) which marks each node in a TPS with a Boolean flag "bounded". The algorithm may traverse a node several times. Therefore we assume that the "bounded" flag is initially true. It may be changed to $false$ once and then remains $false$.*

$\mathcal{BC}$ goes recursively down the tree, changes the "bounded" flags (as a side effect), and returns the convex hull of the reference intervals for the subtrees.

$$\mathcal{BC}((DF, subtrees, continuous)) \stackrel{\text{def}}{=} \mathcal{BC}(subtrees, 0, +\infty, DF, 0)$$

where $\mathcal{BC}((RT_0, \ldots, RT_n), t_0, t_1, DF, level)$ works as follows:

```
For $k = 0, \ldots, n$:
    If $RT_k.subtrees = \emptyset$: let $[c_k, d_k[ \stackrel{\text{def}}{=} NInterval(t_0, t_1, G, G')$
    If $RT_k.subtrees \neq \emptyset$: for all $j \in RT_k.RTNode.NPositions(t_0, t_1, G, G')$
        let $[c_{k,j}, d_{k,j}[ \stackrel{\text{def}}{=} \mathcal{BC}(RT_k.subtrees, a_{k,j}, b_{k,j}, DF, level+1)$
            where $[a_{k,j}, b_{k,j}[ \stackrel{\text{def}}{=} G.nthGranuleI(t_0, j, true, G')$.
            Set $RT_k.bounded = false$ if $c_{k,j} < a_{k,j} \vee b_{k,j} < d_{k,j}$
        Now let $c_k \stackrel{\text{def}}{=} min\{c_{k,j}\}$ and $d_k \stackrel{\text{def}}{=} max\{d_{k,j}\}$
  return $[min\{c_k\}, max\{d_k\}[$
```

We have

$$G \stackrel{\text{def}}{=} DF[level] \text{ and } G' \stackrel{\text{def}}{=} \begin{cases} DF[level-1] & \text{if } level > 0 \\ \emptyset & \text{otherwise.} \end{cases}$$

■

The key function which works at the TPS directly is the function *Part* (Def. 3.3.75 below) which computes for a time point *time* the corresponding partition containing *time*, together with the attached label. The *Part* function goes recursively down the TPS. For each leaf node it computes the partition containing *time*, together with the label attached to it. These partitions get intersected when the recursion goes back to the root nodes.

For a leaf node *RT*, we define a function *RT.PLeaf*, which computes the partition containing *time*, together with the label for this partition.

Definition 3.3.74 (Partition for Leaf Nodes)
For a leaf node $RTN = (NPositions, label, granuleLabel)$ of a TPS, a time point time, a time interval $[t_0, t_1[$, two granularities G and G' and a Boolean flag continuous, we define a function

$$RTN.PLeaf(time, t_0, t_1, G, G', continuous) \overset{\text{def}}{=} \begin{cases} (]-\infty, +\infty[, \emptyset) & \text{if } RTN.NInterval(t_0, t_1, G, G') = \emptyset \\ ([a, b[, label) & \text{otherwise} \end{cases}$$

where
1. $[s, e[\overset{\text{def}}{=} RTN.NInterval(t_0, t_1, G, G')$ *(Def. 3.3.64)*
2. **If** $continuous = true$ **then**

$$[a, b[\overset{\text{def}}{=} \begin{cases}]-\infty, s[& \text{if } time < s \\ [s, +\infty[& \text{otherwise} \end{cases}$$

and

$$label \overset{\text{def}}{=} \begin{cases} \emptyset & \text{if } time < s \\ RT.RTNode.Label(s, G) & \text{otherwise} \end{cases}$$

3. **If** $continuous = false$ **then**

$$[a, b[\overset{\text{def}}{=} \begin{cases}]-\infty, s[& \text{if } time < s \\ [s, e[& \text{if } s \le time < e \\ [e, +\infty[& \text{otherwise} \end{cases}$$

and

$$label \overset{\text{def}}{=} \begin{cases} \emptyset & \text{if } time < s \vee e \le time \\ RT.RTNode.Label(s, G) & \text{otherwise} \end{cases}$$

■

The recursive *Part* function (Def. 3.3.75 below) computes for a time point *time* a tuple $([a, b[, L)$ where $[a, b[$ is the partition containing *time* and L is the corresponding label (if there is one). At each level of the TPS, were $RT_0, \ldots, RT_n$ are the sibling nodes, it does the following:

1. it calls *RT.PLeaf* to collect the partitions for all leaf nodes among $RT_0, \ldots, RT_n$;
2. it calls itself recursively for all unbounded non-leaf nodes to collect the partitions for these nodes;
3. it goes over all bounded non-leaf nodes whose reference granules contain *time* and calls itself recursively;
4. if the previous list is empty, it locates for *time* the neighbouring reference granules and calls itself recursively for these ones.

The result is a list of partition-label pairs $([a_i, b_i[, L_i)$, where all partitions contain *time*. These are intersected to get an interval $[a, b[$. The "closest match" to get the corresponding label L is computed as follows:
If $[a, b[= [a_i, b_i[$ for some i then $L \overset{\text{def}}{=} L_i$. (If there are several such identical intervals then the choice is random).
If $a = a_i$ for some i then the label L_j of the smallest partition $[a_i, b_j[$ is chosen. (If there are several such identical intervals then the choice is random.).

Definition 3.3.75 (The *Part* function) *For a Tree Partitioning Specification $TPS = (DF, subtrees, continuous)$ and a time point time we define a function*

$$Part(TPS, time) \overset{\text{def}}{=} Part(subtrees, time, 0, +\infty, DF, 0)$$

where

$$Part((RT_0,\dots,RT_n),time,t_0,t_1,DF,level) \overset{\text{def}}{=} (([a,b[,L)$$

and $[a,b[$, *and* L *are computed in the following steps:*

1. $G \overset{\text{def}}{=} DF[level]$ *and* $G' \overset{\text{def}}{=} \begin{cases} DF[level-1] & \textit{if } level > 0 \\ \emptyset & \textit{otherwise} \end{cases}$

2. $[s_{i,j},e_{i,j}[\overset{\text{def}}{=} G.nthGranuleI(t_0,k_{i,j},true,G')$ *where* $(k_{i,0},\dots,k_{i,n_i}) \overset{\text{def}}{=} RT_i.RTNode.NPositions(t_0,t_1,G,G')$ *(Def. 3.3.64)*

3. $P_{leaf} \overset{\text{def}}{=} \{RT_i.PLeaf(time,t_0,t_1,G,G',continuous) \mid RT_i \textit{ is a leaf node}\}$ *(Def. 3.3.74)*

4. $P_{unbounded} \overset{\text{def}}{=} \{Part(RT_i.subtrees,time,s_{i,j},e_{i,j},DF,level+1) \mid RT_i \textit{ is an unbounded non-leaf node}, j=0,\dots,n_i\}$

5. $P_{bounded,inside} \overset{\text{def}}{=} \{Part(RT_i.subtrees,time,s_{i,j},e_{i,j},DF,level+1) \mid RT_i \textit{ is a bounded non-leaf node}, s_{i,j} \le time < e_{i,j}\}$

6. $P_{bounded,<} \overset{\text{def}}{=} \{Part(RT_i.subtrees,time,s_{i,j},e_{i,j},DF,level+1) \mid RT_i \textit{ is a bounded non-leaf node}, e_{i,j} < time \wedge \neg\exists e_{i',j'} : e_{i,j} < e_{i',j'} < time \wedge RT_{i'} \textit{ is bounded}\}$

7. $P_{bounded,>} \overset{\text{def}}{=} \{Part(RT_i.subtrees,time,s_{i,j},e_{i,j},DF,level+1) \mid RT_i \textit{ is a bounded non-leaf node}, time \le s_{i,j} \wedge \neg\exists s_{i',j'} : time \le s_{i',j'} < s_{i,j} \wedge RT_{i'} \textit{ is bounded}\}$

8. $P_{bounded,outside} \overset{\text{def}}{=} \begin{cases} \emptyset & \textit{if } P_{bounded,inside} \neq \emptyset \\ P_{bounded,<} \cup P_{bounded,>} & \textit{otherwise} \end{cases}$

9. $P \overset{\text{def}}{=} P_{leaf} \cup P_{unbounded} \cup P_{bounded,inside} \cup P_{bounded,outside}$. *(Def. 3.3.68)*
   Let $P = \{(([a_0,b_0[,L_0),\dots,([a_m,b_m[,L_m)\}$.

10. $a \overset{\text{def}}{=} max\{a_i \mid i=0,\dots,m\}$
    $b \overset{\text{def}}{=} min\{b_i \mid i=0,\dots,m\}$.

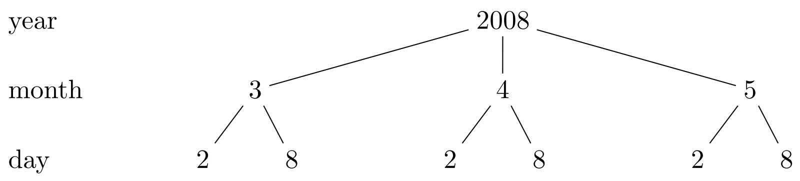
11. **If** $[a,b[\overset{\text{def}}{=} [a_{i_0},b_{i_0}[= \dots = [a_{i_k},b_{i_k}[$ **then** *choose* $L = L_{i_j} \neq \emptyset$ *at random, if there is one, otherwise* $L \overset{\text{def}}{=} \emptyset$.
    Otherwise if $a = a_{i_0} \dots = a_{i_k}$ **then** *choose some* $L \overset{\text{def}}{=} L_{i_j} \neq \emptyset$ *at random with* $b_{i_j} - a_{i_j}$ *is minimal for* $j=0,\dots,k$. *If there is no such* L_{i_j} *then let* $L \overset{\text{def}}{=} \emptyset$. ■

Definition 3.3.76 (Generated Tree Partitioning) *A Tree Partitioning Specification* $TPS \overset{\text{def}}{=} (DF, subtrees, continuous)$ *generates the following labelled partitioning* $P = (IS_P, sPa)$ *(Def. 3.3.4) where*

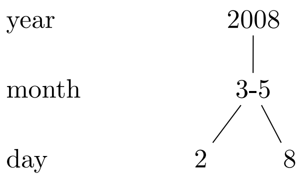
1. $sPa((P,lower,upper,l)) \overset{\text{def}}{=} lower$ *and*

2. $P.PaCo(time) \overset{\text{def}}{=} (TPS,a,b,L)$ *where* $([a,b[,L) \overset{\text{def}}{=} Part(TPS,time)$ *(Def. 3.3.75)* ■

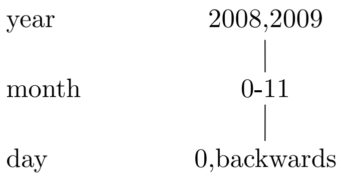
Range Tree Node Types. We now consider Range Tree Node types and the corresponding *NPositions* function in more detail. Conceptually, Range Tree Node Types consist of a list of relative granule positions. The tree



shows a very simple example where all Range Tree Nodes contain a single relative granule position. The tree is very redundant because all three subtrees of the month nodes are identical. It would be much more convenient for the user if he could just specify



where "3-5" represents the list 3,4,5. This is an example where "3-5" just saves some writing. There are, however, examples where the relative granule positions become context dependent. Consider the specification "in the years 2008 and 2009, in every month, in the last day of the month". The corresponding tree could be:



The flag "backwards" indicates that the relative day position is to be calculated backwards from the end of the month. "0,backwards" means the last day of the month, which, of course, is different for the different months and for leap years and non-leap years. It is the purpose of the *NPositions*-functions to compute, possibly context dependent, from compact specifications like "0,backwards" the corresponding relative granule positions.

The definitions 3.3.77, 3.3.78 3.3.79 3.3.80 below introduce four different kinds of Range Tree Node types. They allow for a very convenient specification of relative granule position. The corresponding *NPositions*-functions turn the convenient specifications into a list of relative granule positions.

Number Range Nodes. The first Range Tree Node type, Number Range RTNodes, can be used to list just intervals of relative granule positions. It has two flags, $forward$ and *bounded*. $forward = true$ means that the granule positions are counted from past to the future, and $forward = false$ means that they are counted from future to past. $bounded = true$ means that, if the listed positions are larger that the actual number of positions in the given context, they are automatically restricted to the allowed positions in the context. $bounded = false$ causes no automatic restriction. We can illustrate this with the following examples:

$$\begin{tikzcd}
\text{year} & \text{2008,2009} \arrow[d, no head] \\
\text{week} & \text{50-52,forward,bounded}
\end{tikzcd}$$

The year 2008 has 52 weeks (week 0 - 51) and the year 2009 has 53 weeks (week 0 - 52). If $forward = true$ and $bounded = false$ then $50 - 52$ is expanded to 50,51,52 in both years. The week 52 in the year 2008 is then actually the first week in 2009. If $forward = true$ and $bounded = true$ then $50 - 52$ is expanded to 50,51 for the year 2008 and 50,51,52 for the year 2009. If $forward = false$ and $bounded = false$ then $50 - 52$ is counted from backwards. Week 0, backwards, is the last week in the year. $50 - 52$ is then expanded to 1,0,-1 for the year 2008 and 2,1,0 for the year 2009. Week -1 for the year 2008 is the last week in 2007. If $forward = false$ and $bounded = true$ then all negative granule positions are deleted. $50 - 52$ is then expanded to 1,0 for the year 2008 and 2,1,0 for the year 2009.

Notice that $bounded = true$ causes a restriction only in one direction. For example, week $-2 - +2$ with $forward = true$ and $bounded = true$ expands to -2,-1,0,1,2, i.e. $bounded = true$ has no effect. $-2 - +100$ with $forward = true$ and $bounded = true$, however, expands in the year 2008 to all numbers between -2 and 51. Thus, $bounded = true$ has an effect in the forward direction. Week $-2 - +2$ with $forward = false$ and $bounded = true$ expands in the year 2008 to 53,52,51,50,49. Again, $bounded = true$ has no effect. For week $-2 - +100$ with $forward = false$, $bounded = true$ has the effect to restrict the numbers to 53,52,...,0 in the year 2008.

Definition 3.3.77 (Number Range RTNodes)
A Number Range RTNode $NRN = ((a_0, b_0), \ldots, (a_n, b_n), forward, bounded)$ consists of a list of pairs (a_i, b_i) of integers with $a_i \leq b_i$ for $i = 0, \ldots, n$ and two Boolean flags, forward and bounded. The integers determine relative granule positions. $forward = true$ means that the relative granule positions are counted from past to future. $forward = false$ means that the relative granule positions are counted from future to past. $bounded = true$ causes the NPositions function (see below) to restrict the relative granule positions in counting direction to the given time interval.

A Number Range RTNode generates the following NPositions function:

$$NRN.NPositions(t_0, t_1, G, G') = N'$$

where
1. $[t_0, t_1[$ *is a non-empty time interval,*
2. G *and* G' *are granularities (G' is irrelevant here)*
3. $P \overset{\text{def}}{=} G.partitioning$
4. $(s_0, e_0) \overset{\text{def}}{=} G.firstGranule(t_0)$
5. $(s_1, e_1) \overset{\text{def}}{=} G.firstGranule(t_1)$
6. $d \overset{\text{def}}{=} \lfloor G.lengthG(P.ePa(e_0), P.sPa(s_1)) \rfloor$ *(Def. 3.3.32)*
7. $N \overset{\text{def}}{=} \begin{cases} (a_0, \ldots, b_0, \ldots, a_n, \ldots, b_n) & \textit{if forward} \\ (d - a_0, \ldots, d - b_0, \ldots, d - a_n, \ldots, d - b_n) & \textit{otherwise} \end{cases}$
8. $N' \overset{\text{def}}{=} \begin{cases} \{i \in N \mid i \leq d\} & \textit{if bounded} \wedge forward \\ \{i \in N \mid i \geq 0\} & \textit{if bounded} \wedge \neg forward \\ N & \textit{otherwise} \end{cases}$

■

Number Iterator Nodes. Number Range RTNodes are no longer convenient when the relative granule positions have a regular pattern. For example "every second week" could be expressed with a Number Range Node, but the weeks need to be listed explicitely. The next node type, Number Iterator RTNodes, provides 'for-loop' like functionality to simplify the specification of granule positions with regular patterns. The following example illustrates this.

```
year                2008,2009
                        |
week        3,2,30,forward,bounded
```

The week node is now a Number Iterator RTNode. The 2 means, 'every second week', the 3 means, 'starting from week 3' and the 30 means, '30 iterations'. $forward$ and *bounded* have the same meaning as in Number Range RTNodes. If $forward = true$ and $bounded = false$ then 3,2,30 is expanded to the weeks 3,5,7,...,61. If $forward = true$ and $bounded = true$ then 3,2,30 is expanded to the weeks 3,5,7,...,51 for both years, 2008 and 2009. The weeks from 53 onwards would be outside both years. They are therefore deleted. If $forward = false$ and $bounded = false$ then 3,2,30 is expanded in the year 2008 to the weeks 48,46,...,-10 (week 0, backwards is week 51, therefore week 3, backwards is week 48). In 2009 it is expanded to the weeks 49,47,...,-9. If $forward = false$ and $bounded = true$ then 3,2,30 is expanded in the year 2008 to the weeks 48,46,...,0 and in 2009 to the weeks 49,47,...,+1.

Definition 3.3.78 (Number Iterator RTNodes)
A Number Iterator RTNode $NIN = (start, step, iterations, forward, bounded)$ *consists of three integers start, step and iterations and two Boolean flags* $forward$ *and bounded. where* $start \geq 0$, $step > 0$ *and* $iterations > 0$.

start, step and iterations enumerate a sequence of integers like in a for loop. The integers determine relative granule positions. $forward = true$ *means that the relative granule positions are counted from past to future.* $forward = false$ *means that the relative granule positions are counted from future to past.* $bounded = true$ *causes the NPositions function (see below) to restrict the relative granule positions in the direction of the iteration to the given time interval.*

A Number Iterator RTNode generates the following NPositions function:

$$NIN.NPositions(t_0, t_1, G, G') = N'$$

where
1. $[t_0, t_1[$ *is a non-empty time interval,*
2. G *and* G' *are granularities* (G' *is irrelevant here)*
3. $P \overset{\text{def}}{=} G.partitioning$
4. $(s_0, e_0) \overset{\text{def}}{=} G.firstGranule(t_0)$
5. $(s_1, e_1) \overset{\text{def}}{=} G.firstGranule(t_1)$
6. $d \overset{\text{def}}{=} \lfloor G.lengthG(P.ePa(e_0), P.sPa(s_1)) \rfloor$
7. $a_i \overset{\text{def}}{=} \begin{cases} start + i \cdot step & \text{if } forward \\ d - (start + i \cdot step) & \text{otherwise} \end{cases}$
8. $N \overset{\text{def}}{=} \{a_0, \ldots, a_{iterations}\}$
9. $N' \overset{\text{def}}{=} \begin{cases} \{i \in N \mid i \leq d\} & \text{if } bounded \wedge forward \\ \{i \in N \mid i \geq 0\} & \text{if } bounded \wedge \neg forward \\ N & \text{otherwise} \end{cases}$

■

Label Range Nodes. Sometimes it may be more convenient to specify relative granule positions not as numbers, but as labels. As an example, suppose we have the days labelled Mo,Tu,We,Th,Fr,Sa,Su and a label hierarchy LH, which lists the labels Mo,Tu,We,Th,Fr as sub-labels of the label WD (working day). A specification "in the year 2008, every working day and every Sunday" could be represented:

$$\begin{tikzcd} \text{year} & 2008 \arrow[d, dash] \\ \text{day} & \{\text{WD,Su}\},\text{LH} \end{tikzcd}$$

The day node is now a Label Range Node. It expands to the relative day positions: 0,1,2,3,5,6,7, ..., 365 (day 0 in 2008 is a Tuesday, day 3 is a Friday, day 5 is a Sunday, day 365 is a Wednesday (2008 is a leap year)).

Definition 3.3.79 (Label Range RTNodes)
A Label Range RTNode $LRN = (L, LH)$ consists of a set L of labels and a label hierarchy LH (Def. 3.2.2). The labels determine relative granule positions of the granules labelled with some label in the given set, or a sub-label in the label hierarchy.

A Label Range RTNode generates the following NPositions function:

$$LRN.NPositions(t_0, t_1, G, G') = G.granulesWithLabel(t_0, t_1, L, LH, G')$$

where
1. $[t_0, t_1[$ is a non-empty time interval,
2. G and G' are granularities ■

Label Iterator Nodes. These nodes are similar to Number Iterator Nodes (Def. 3.3.78). Instead of numbers, however, it iterates over label occurrences. Consider again the example, where the days are labelled Mo,Tu,We,Th,Fr,Sa,Su and the label hierarchy LH lists the labels Mo,Tu,We,Th,Fr as sub-labels of the label WD (working day). A specification with a Label Iterator Node could be:

$$\begin{tikzcd} \text{year} & 2008 \arrow[d, dash] \\ \text{day} & \text{WD,LH,3,2,30,forward,bounded} \end{tikzcd}$$

The 2 means 'every second occurrence of a sub-label of WD', i.e. every second working day. The 3 means, 'starting with occurrence number 3'. The 30 means, 'at most 30 of them'. Notice that there is no *bounded* flag because the label occurrences are automatically restricted to the given intervals (the year 2008 in this example).

If $forward = true$ then the day node is expanded to 3,7,9,13,.... Day 3 is Friday, 4th of January. There are 3 working days before this Friday in 2008. Day 7 is the following Tuesday. One working day is between day 3 and day 7.

If $forward = false$ then the day node is expanded to 360,358,356,352,.... Day 360 is Friday, 26th of December. There are three working days, Monday - Wednesday between this Friday and the end of the year. Day 358 is a Wednesday, day 352 is Monday and day 352 is a Thursday. The *bounded* flag causes the relative granule positions to be restricted to the reference interval (the year 2008 in this example).

Definition 3.3.80 (Label Iterator RTNodes)
A Label Iterator RTnode $LIN = (l, LH, start, steps, iterations, forward, bounded)$ *consists of a label* l*, a label hierarchy* LH*, three numbers start, steps and iterations and two Boolean flags* $forward$ *and bounded.*

start, step and iterations enumerate a sequence of integers like in a for loop. The integers represent relative granule positions of the granules labelled with l *or a sub-label in the hierarchy* LH*.* $forward = true$ *means that the relative granule positions are counted from past to future.* $forward = false$ *means that the relative granule positions are counted from future to past.* $bounded = true$ *causes the relative granule positions to be restricted to the reference interval.*

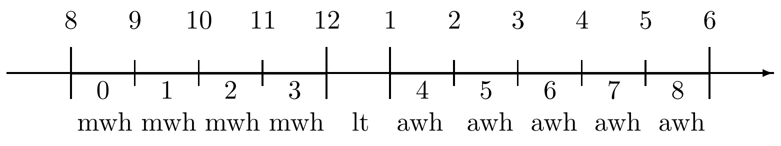
A Label Iterator RTNode generates the following NPositions function:

$$LIN.NPositions(t_0, t_1, G, G') = N$$

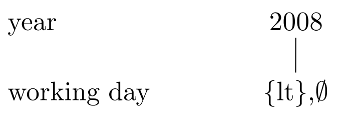
where

1. $[t_0, t_1[$ *is a non-empty time interval,*
2. G *and* G' *are granularities,*
3. $(s_0, e_0) \overset{\text{def}}{=} G.firstGranule(t_0)$,
4. $(s_1, e_1) \overset{\text{def}}{=} G.firstGranule(t_1)$,
5. $N \overset{\text{def}}{=} (i_0, \ldots, i_n)$ *is a sorted subset of* $G.granulesWithLabel(t_0 - \alpha, t_1 + \beta, \{l\}, LH, G')$ *with*
   $|G.granulesWithLabel(i_k, i_{k+1}, \{l\}, LH, G')| = steps - 1$ *for all* $k = 0, \ldots n - 1$ *and*
   $|G.granulesWithLabel(s_0, i_0, \{l\}, LH, G')| = start$ *if* $forward$ *and*
   $|G.granulesWithLabel(i_n, s_1, \{l\}, LH, G')| = start$ *if* $\neg forward$
   and $n \leq iterations - 1$ *is maximal such that the above conditions are met, and where*
   $\alpha = \beta = 0$ *if* $bounded = true$ *and*
   $\alpha = 0$ *and* β *is large enough such that* $|N| = iterations$ *or there are no granule positions beyond* i_n *with the given label, if* $bounded = false$ *and* $forward = true$*, and*
   $\beta = 0$ *and* α *is large enough such that* $|N| = iterations$ *or there are no granule positions before* i_0 *with the given label, if* $bounded = false$ *and* $forward = false$*.* ■

Both, Label Range Nodes and Label Iterator Nodes, specify relative *granule* positions, by referring to labels attached to partitions. This is very natural if granules and partitions are the same, as in the 'day' example, above. Identifying more complex granules by a single label, however, may be convenient, but it may also yield quite strange results. Consider our 'working day' granularity again.



The specification



would enumerate all working day granularities in the year 2008 by referring to the label *lt* of the lunch time. The specification

$$\begin{tikzcd} \text{year} & 2008 \arrow[d, dash] \\ \text{working day} & \{\text{lt,mwh}\},\emptyset \end{tikzcd}$$

yields the same enumeration of all working day granularities, although one of the labels lt or mwh is redundant here.

3.3.9 Calendar Systems

A large part of PartLib deals with the definition of calendar systems. First of all, PartLib distinguishes individual calendar systems and calendar sequences. Individual calendar systems are ordinary calendar systems like the Gregorian calendar. Calendar sequences combine sequences of calendar systems in one composed system. An example could be the Julian calendar system followed in 1582 by the Gregorian system. Another calendar sequence could be the sequence of calendar systems a traveller passes on a trip around the world.

Individual calendar systems are essentially a collection of granularities like years, months etc. They may have one or more predefined date formats, for example, year/month/day hour:minute:second and year/week/day hour:minute:second. They have methods for parsing dates in different formats and generating date strings. Each of the component granularities offers of course all the functionalities, a granularity can offer in PartLib.

Individual calendar systems can be composed of separately definable components:

- Sub-Seconds.
  These can be fractions of a second, for example milliseconds, microseconds etc.
- Hour-Minute-Second,
  which are the usual components, hours minutes, seconds;
- Year-Month-Week-Day.
  This is a general class which can be refined with the details of various calendar systems (see [13]).

Within the Calendar System part of PartLib there is no support for time units like decades or centuries. These can be defined as ordinary partitionings.

There is an XML-specification for composing application specific calendar systems from the built-in components. For example, one could define a particular calendar system consisting of the Julian Year-Month-Week-Day component, the ordinary Hour-Minute-Second component together with milliseconds and microseconds and place it into a particular time zone.

3.3.10 The XML-Interface

This section is yet to be included.

Chapter 4

GeTS– A Specification Language for Geo-Temporal Notions

Abstract: This document describes the 'Geo-Temporal' specification language GeTS. The objects which can be described and manipulated with this language are time points, crisp and fuzzy time intervals and labelled partitionings of the time axis. The partitionings are used to represent periodic temporal notions like months, semesters etc. GeTS is essentially a typed functional language with a few imperative constructs. GeTS can be used to specify and compute with many different kinds of temporal notions, from simple arithmetic operations on time points up to complex fuzzy relations between fuzzy time intervals. The syntax of GeTS together with an operational semantics is described. A parser, a compiler and an abstract machine for GeTS is implemented. The application programming interface for GeTS is documented in the appendix.

4.1 The GeTS Language

The design of the GeTS language was influenced by the following considerations:

1. Although the GeTS language has many features of a functional programming language, it is not intended as a general purpose programming language. It is a specification language for temporal notions, however, with a concrete operational semantics.

2. The parser, compiler, and in particular the underlying GeTS abstract machine are not standalone systems. They must be embedded into a host system which provides the data structures and algorithms for time intervals, partitionings etc., and which serves as the interface to the application. GeTS provides a corresponding application programming interface (API).

3. The language should be simple, intuitive, and easy to use. It should not be cluttered with too many features which are mainly necessary for general purpose programming languages.

4. The last aspect, but even more the point before, namely that GeTS is to be integrated into a host system, were the main arguments against an easy solution where GeTS is only a particular module in a functional language like SML or Haskell. The host system was developed in C++. Linking a C++ host system to an SML or Haskell interpreter for GeTS would be more complicated than developing GeTS in C++ directly. The drawback is that features like sophisticated type inferencing or general purpose data structures like lists or vectors are not available in the current version of GeTS. If it turns out that they are useful for some applications, however, it is not a big deal to integrate them into GeTS.

5. Developing GeTS from scratch instead of using an existing functional language has also an advantage. One can design the syntax of the language in a way which better reflects the semantics of the language constructs. This makes it easier to understand and use. As an example, the syntax for a time interval constructor is just $[expression_1, expression_2]$. The freedom in designing a nice syntax is, however, is limited by the available parser technology (in the GeTS case, flex and bison). Therefore some of the language features are compromises between intuitiveness and technical constraints.

The GeTS language is a strongly typed functional language with a few imperative constructs. Let us get a flavour of the language, before the technical details are introduced.

Example 4.1.1 (tomorrow) The definition

```
tomorrow = partition(now(),day,1,1)
```

specifies 'tomorrow' as follows: `now()` yields the time point of the current point in time (Def. 4.1.25). `day` is the name of the day partitioning. Let i be the coordinate of the day-partition containing `now()`. `partition(now(),day,1,1)` computes the interval $[t_1, t_2[$ where t_1 is the start of the partition with coordinate $i + 1$ and t_2 is the end of the partition with coordinate $i + 1$. Thus, $[t_1, t_2[$ is in fact the interval which corresponds to 'tomorrow'.

In a similar way, we can define

```
this_week(Time t)  = partition(t,week,0,0).
```

The time point `t`, for which the week is to be computed, is now a parameter of the function. ■

Example 4.1.2 (Christmas) The definition

```
christmas(Time t) =
  dLet year = date(t,Gregorian_month) in
                 [time(year|12|25,Gregorian_month),
                  time(year|12|27,Gregorian_month)]
```

specifies Christmas for the year containing the time point `t`. ■

`date(t,Gregorian_month)` computes a date representation for the time point `t` in the date format `Gregorian_month` (`year/month/day/hour/minute/second`). Only the year is needed. `dLet year = ...` therefore binds only the year to the integer variable `year`. If, for example, in addition the month is needed one can write `dLet year|month = date(...`.

`time(year|12|25,Gregorian_month)` computes t_1 = begin of the 25th of December of this year. `time(year|12|27,Gregorian_month)` computes t_2 = begin of the 27th of December of this year. The expression `[...,...]` denotes the half-open interval $[t_1, t_2[$.[1] The result is therefore the half-open interval from the beginning of the 25th of December of this year until the end of the 26th of December of this year.

Example 4.1.3 (Point–Interval Before Relation) The function

```
PIRBefore(Time t, Interval I) =
     if (isEmpty(I) or isInfinite(I,left)) then false
     else (t < point(I,left,support))
```

specifies the standard crisp point–interval 'before' relation in a way which works also for fuzzy intervals. ■

If the interval `I` is empty or infinite at the left side then `PIRBefore(t,I)` is `false`, otherwise `t` must be smaller than the left boundary of the support of `I`.

Now we define a parameterized fuzzy version of the interval–interval before relation.

Example 4.1.4 (Fuzzy Interval–Interval Before Relation) A fuzzy version of an interval–interval before relation could be

[1]Crisp intervals in CTTN are always half-open intervals $[\ldots, \ldots[$. Sequences of such intervals, for example sequences of days, can therefore be used to partition a time period. The syntactic representation of these intervals in GeTS is `[...,...]` and not `[...,...[` because this simplifies the grammar and the parser considerably.

```
 IIRFuzzyBefore(Interval I, Interval J, Interval->Interval B) =
   case
     isEmpty(I) or isEmpty(J) or isInfinite(I,right) or isInfinite(J,left) : 0,
     (point(I,right,support) <= point(J,left,support))                     : 1,
      isInfinite(I,left) : integrateAsymmetric(intersection(I,J),B(J))
   else integrateAsymmetric(I,B(J))
```

■

The input are the two intervals `I` and `J` and a function `B` which maps intervals to intervals. `B` is used to compute for the interval `J` an interval `B(J)`, which represents the degree of 'beforeness' for the points before `J`.

The function first checks some trivial cases where `I` cannot be before `J` (first clause in the `case` statement), or where `I` definitely is before `J` (second clause in the `case` statement). If `I` is infinite at the left side then $\int (I \cap J)(x) \cdot B(J)(x)dx/|I \cap J|$ is computed to get a degree of 'beforeness', at least for the part where I and J intersect. If `I` is finite then $\int I(x) \cdot B(J)(x)dx/|I|$ is computed. This averages the degree of a point-interval 'beforeness', which is given by the product $I(x) \cdot B(J)(x)$, over the interval `I`.

The next example illustrates some procedural features of GeTS. The `effect` function takes two intervals and a function `F`, which maps the two intervals to a fuzzy value. `F` could for example be a fuzzy interval–interval relation. The first interval `I` is now shifted `step` times by the given `distance`, and each time `F(I,J)` is computed. These values are inserted into a new interval, which is the result of the function.

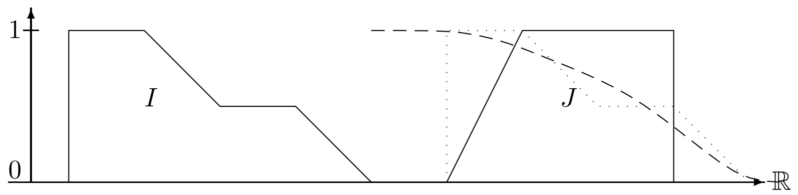
Example 4.1.5 (effect)

```
effect(Interval I, Interval J, (Interval*Interval)->Float F,
       Time distance, Integer steps) =
      Let K = [] in
            while (steps >= 0) {
                  pushBack(K,point(I,right,kernel),F(I,J)),
                  I := shift(I,distance),
                  steps := steps - 1}
            K
```

■

'`Let K = []`' creates a new empty interval and binds it to the variable `K`. The `while` loop shifts the interval `I` `steps` times by the given `distance` (`I := shift(I,distance)`). Each time `pushBack(K,point(I,right,kernel),F(I,J))` adds the pair (x, y) consisting of x = right boundary of the kernel of the shifted `I` and $y =$ `F(I,J)` to the interval `K`.

The dashed line in the figure below shows the result of the `effect` function when applied to the two intervals `I` and `J`, and a suitable interval–interval 'before' relation as parameter `F`. The dotted figure shows the position of the shifted interval `I` when the `F(I,J)` drops down to 0.



Effect of the `effect` *function*

4.1.1 Types in the GeTS Language

The GeTS language has a fixed number of basic types. They represent certain data structures and certain keywords. So far there is no mechanism for extending the basic types. The basic types can be combined to functional types $T_1 * \ldots * T_n \mapsto T$.

4.1.1.1 Basic Types

There are two groups of basic types, the data structure types and the enumeration types. The data structure types represent built in data structures.

Definition 4.1.6 (Data Structure Types)

`Integer`	*standard integers*
`Time`	*very long integers*
$\mathbb{F}$loat	*standard floating point numbers*
`String`	*strings*
`Interval`	*fuzzy intervals*
`Partitioning`	*partitionings*
`Label`	*labels for partitions*
`Duration`	*durations*
`DateFormat`	*date formats*

■

The data structure types abstract away from the concrete implementation. The `Integer` type, for example, is currently realized as 32 bit signed integer data, while the `Time` type is currently realized as 64 bit signed integer data. The $\mathbb{F}$loat type is currently realized by a 32 bit 'float' data type. One should, however, not exploit this in any way.

`String`s are currently just sequences of 8-bit characters. (This may change in future releases to support Unicode).

`Interval`s are realized as *polygons* with integer coordinates. An interval is therefore a sequence of pairs $I = (x_0, y_0), \ldots, (x_n, y_n)$. The x_i are `Time` points and the y_i are fuzzy values. Internally the y_i are realized as short integers between 0 and 1000. From the GeTS point of view, however, the y_i are $\mathbb{F}$loat numbers between 0 and 1. The interval I is *negative infinite* if $y_0 \neq 0$. I is *positive infinite* if $y_n \neq 0$. The internal representation of `Interval` data, however, is completely invisible to the GeTS user. Details about the internal representation and the algorithms can be found in [31].

`Partitioning`s are complex data structures. Fortunately, this is also not visible to the GeTS user. Partitionings are just parameters to some of the functions. They can be used without knowing anything about the internal details.

`Label`s for partitions are in principle just strings. It is, however, possible to use different strings for the same label. For example, one can label days with English names "Monday", "Tuesday" etc., and with German names "Montag", "Dienstag" etc., and switch between these versions. This is also transparent to the user. Nevertheless, it makes it necessary to consider labels not as strings, but as data structures (see [33] for details).

`Duration`s (Def. 3.3.37) are sequences of pairs $d_0\ P_0, \ldots, d_n\ P_n$ where the d_i are $\mathbb{F}$loat data and the P_i are *Partitionings*.

`DateFormat`s (Def. 3.3.48) are sequences $P_0/\ldots/P_n$ of Partitionings.

The data structure types are used as types for variables, but they can also be used explicitly as constants, so called literals. To this end, there is a string representation of the data structure types. These strings are parsed by the GeTS parser and mapped to the internal representation.

Remark 4.1.7 (String Representation of Data Structure Types) The data structure types have the following string representation:

`Integer`**:** sequences of digits, optionally preceded by '+' or '-'. Examples are 123, +4, -345. The length of these sequences depends on the internal representation of integers.

`Time`**:** sequences of digits, optionally preceded by '+' or '-' and optionally followed by 'T'. Examples: 12345678901, 3T, -23T. The length of these sequences depend on the internal representation of `Time` values.

Notice that a string of digits, which is not followed by 'T' is first parsed as `Integer`. Only if this fails, it is parsed as `Time` data. Therefore the string 123 will always be mapped to `Integer` data, and not to `Time` data. Usually this should not harm, because `Integer` data are automatically casted to `Time` data when this becomes necessary.

$\mathbb{F}$loat**:** They have the standard representation of float or double values. Examples are -1.5, 3.4e-2, -77e+5. The length of mantissa and exponent depends on the realization of $\mathbb{F}$loat values.

`String`**:** They are arbitrary sequences of characters enclosed in quotes: "characters". The two characters \n are interpreted as newline command. A quote " within the string must be escaped with a \ character. The character sequences "ab\"cd\"ef" is therefore parsed as the string ab"cd"ef.

`Interval`**:** Intervals cannot be explicitly referenced within a GeTS function definition. The only exception is the empty interval, which is represented by []. The GeTS module, however, provides an interface function which allows one to call GeTS functions with a string representation of the arguments. This function accepts non-negative integers as identifiers for the intervals, together with a vector of pointers to the actual intervals. The integer identifiers are used as indices to this vector.

`Partitioning`**:** It is assumed that all necessary partitionings are predefined, and can be identified by *names*. The names can be used in GeTS function definitions. The parser maps them to the actual partitioning data structures. The names of the partitionings, which make up the Gregorian calendar, are `year`, `month`, `week`, `day`, `hour`, `minute`, `second`.

`Label`**:** Labels must also be predefined. They are identified by their name.

`Duration`**:** The representation of a *simple* duration is $d_0\ P_0 + \ldots + d_n\ P_n$ where the d_i are `Integer` or $\mathbb{F}$loat expressions and the P_i are partitioning names or variables. Repeating patterns like '2 week + 1 day + 2 week + 1 day' can be abbreviated by '2*(2 week + 1 day)'.

`DateFormat`**:** They need to be predefined. Date formats are accessed by their names in GeTS specifications. Date formats in CTTN are predefined for each calendar system. The predefined date formats for the Gregorian calendar are `Gregorian_month` and `Gregorian_week`. Thus, GeTS invokes calendar systmes, but only implicitely via the data formats.

■

A number of enumeration types is predefined in GeTS. They are used to control some of the algorithms. Their meaning therefore depends on the meaning of the built-in function where they occur as parameters.

Definition 4.1.8 (Enumeration Types)

type name	*possible values*
`Bool`	`true/false`
`Side`	`left/right`
`PosNeg`	`positive/negative`
`UpDown`	`up/down`
`IntvRegion`	`core/kernel/support`
`PointRegion`	`core/kernel/support/maximum`
`Hull`	`core/kernel/support/crisp/monotone/convex`
`Fuzzify`	`linear/gaussian`
`Inclusion`	`subset/overlaps/bigger_part_inside`
`SplitInclusion`	`align/subset/overlaps/bigger_part_inside`
`Sequencing`	`sequential/overlapping/with_gaps`
`SDVersion`	`Kleene/Lukasiewicz/Goedel`

■

Notice that, for example, the keyword `core` occurs in the enumeration types `IntvRegion`, `PointRegion` and `Hull`. Which type is meant is determined by the context where it occurs. If the context is not clear, for example in comparisons 'expression == keyword', one can use `Icore` (type `IntvRegion`), `Pcore` (type `PointRegion`), or `Hcore` (type `Hull`). The same holds of the keywords `kernel` and `support`.

An unknown string is parsed in the following way:
1. is it an `Integer` value?
2. is it a `Time` value?
3. is it a $\mathbb{F}$loat value?
4. is it a keyword of one of the enumeration types?
5. is it a partitioning?
6. is it a date format?

If none of these succeed then a parse error is generated.

Definition 4.1.9 (Basic Types) *A Basic Type in GeTS is either a* data structure type *(Def. 4.1.6), an* enumeration type *(Def. 4.1.8), or the special type* `Void` *for expressions which do not return any values.* ■

Automatic Type Conversion:
Automatic type conversion is done from the type `Integer` to the types $\mathbb{F}$loat and `Time`. That means, the type `Integer` is also acceptable whenever a type $\mathbb{F}$loat or a type `Time` is required.

4.1.1.2 Compound Types

Definition 4.1.10 (Compound Type) *A compound type in GeTS is an expression* $T_1 * \ldots * T_n \mapsto T$ *where* T *and the* T_i *are either basic types or compound types.*
A type expression *is either a basic type or a compound type expression.* ■

4.1.2 Language Constructs for GeTS

The GeTS language has a number of general purpose functional and imperative language components. Additionally a number of language constructs are geared to manipulating time points, temporal intervals, partitionings, dates etc. As already mentioned, the language is strongly typed. This means, the type of each expression is determined by the top level function name together with the types of its arguments.

GeTS tries to minimize the required number of parentheses in the expressions. Nevertheless, it is usually clearer and easier to understand when enough parentheses are used.

The language has an operational semantics. It is described more or less formally when the language constructs are introduced. The explanations should be clear enough to understand what the language is able to do.

Some aspects of the language depend on the context where it is used. For example, GeTS itself has no exception mechanisms. Nevertheless, exceptions are thrown and must be catched by the host programming system.

Definition 4.1.11 (Function Definitions) *A GeTS function definition has one of the forms*

$$\begin{array}{lrcl}
(1) & name & = & expression \\
(2) & name() & = & expression \\
(3) & name(type_1\ var_1, \ldots, type_n\ var_n) & = & expression \\
(4) & type : name(type_1\ var_1, \ldots, type_n\ var_n) & = & expression \\
(5) & type : name(type_1\ var_1, \ldots, type_n\ var_n) & &
\end{array}$$

The five versions of function definitions can have a trailer: '`explanation`*: any string'. The explanation is attached at the newly defined function. It can be accessed by the host system.* ■

Version (1) and (2) are for constant expressions, i.e. the name at the left hand side is essentially an abbreviation for the expression at the right hand side. Version (3) is the standard function definition. The type of the function is $type_1 * \ldots * type_n \mapsto T$ where T is the type of the *expression*. Version (4) declares the range type of the function explicitly. It can be used for recursive function definitions, where the name of the newly defined function occurs already in the body. In this case it is necessary to know the range type of the function, before the *expression* can be fully parsed. The factorial function, for example, must be defined in this way:

```
Integer:factorial(Integer n) = if(n == 0) then 1 else n * factorial(n-1)
```
(4.1)

Finally, version (5) is a forward declaration. It must be used for mutually recursive functions.

Remark 4.1.12 (Overloading) *Function definitions can be overloaded. They are distinguished by their argument types, not by the result type. This means, two function definitions*

`f(Integer n) = ...` *and*
`f(𝔽loat m) = ...`

yield different functions, whereas the second definition in

`Integer:f(Integer n) = ...` *and*
`𝔽loat:f(Integer n) = ...`

overwrites the first one or is rejected. This depends on the global control parameter `GeTS::overwrite`*.* ∎

Definition 4.1.13 (Literals) *Literals are strings which can be interpreted as constants of a certain type. See Remark 4.1.7 for the string representation of literals.* ∎

4.1.2.1 Arithmetic Expressions

GeTS supports the same kind of arithmetic expressions as many other programming languages. A small difference is the `Time` type, which is integrated in the arithmetics of GeTS.

Definition 4.1.14 (Binary Arithmetic Expressions)
Let N be a number type (i.e. $N =$ `Integer` or $N =$ 𝔽loat or $N =$ `Time`).
If n and m are valid arithmetic expressions then the following binary operations are allowed:

$n + m$	*(addition)*	$n \% m$	*(modulo)*
$n - m$	*(subtraction)*	$max(n, m)$	*(maximum)*
$n * m$	*(multiplication)*	$min(n, m)$	*(minimum)*
n/m	*(division)*	$pow(n, e)$	*(n^e)*

The types are determined according to the following rules:
for the operators '+', '-', '', '/', max and min:*

`Integer`	$*$	`Integer`	$\mapsto$	`Integer`	`Time`	$*$	`Integer`	$\mapsto$	`Time`
𝔽loat	$*$	`Integer`	$\mapsto$	𝔽loat	`Time`	$*$	`Time`	$\mapsto$	`Time`
`Integer`	$*$	𝔽loat	$\mapsto$	𝔽loat	𝔽loat	$*$	`Time`	$\mapsto$	`Time`
𝔽loat	$*$	𝔽loat	$\mapsto$	𝔽loat	`Time`	$*$	𝔽loat	$\mapsto$	`Time`
`Integer`	$*$	`Time`	$\mapsto$	`Time`					

The last two type patterns mean that the result of operations on mixed 𝔽loat *and* `Time` *values are rounded to* `Time` *values. This makes the operations non-associative:* $1.5 + 1.5 + 1T = 4$, *whereas* $1.5 + 1T + 1.5 = 3$.

𝔽loat *values are not allowed for the modulo operator %. Therefore the remaining type patterns for % are:*

`Integer`	$*$	`Integer`	$\mapsto$	`Integer`	`Time`	$*$	`Integer`	$\mapsto$	`Time`
`Integer`	$*$	`Time`	$\mapsto$	`Time`	`Time`	$*$	`Time`	$\mapsto$	`Time`

The exponentiation operator $pow(n, e)$ is only allowed for `Integer` *exponents and for* 𝔽loat *or* `Integer` *mantissas.*

`Integer`	$*$	`Integer`	$\mapsto$	`Integer`
𝔽loat	$*$	`Integer`	$\mapsto$	𝔽loat.

∎

Flat expressions like $a + b + c + d$ without parentheses are allowed. The operator precedence is -, +, /, *, i.e. * binds most. The functions *min* and *max* can also accept more than two arguments.

Definition 4.1.15 (Unary Arithmetic Expressions) *There are four unary arithmetic operators in GeTS:*

$-n$	$[N \mapsto N]$	*N is any number type*
`float`(b)	$[$`Bool` $\mapsto \mathbb{F}\text{loat}]$	
`round`(a)	$[\mathbb{F}\text{loat} \mapsto$ `Integer`$]$	
`round`$(a,$`up/down`$)$	$[\mathbb{F}\text{loat} *$ `UpDown` $\mapsto$ `Integer`$]$	

■

$-n$ negates the number n.
n can be an expression of type $N =$ `Integer`, $N = \mathbb{F}\text{loat}$ or $N =$ `Time`.

`float`(b) turns a boolean value b into a floating point number:
`float(false)` $= 0.0$ and `float(true)` $= 1.0$.

`round`(a) rounds a $\mathbb{F}$loat value a to the nearest integer. 1.5 is rounded to 1, 1.51 is rounded to 2. -1.5 is rounded to -1, -1.51 is rounded to -2.
`round`$(a,$`up`$)$ rounds the $\mathbb{F}$loat value a up, and
`round`$(a,$`down`$)$ rounds the $\mathbb{F}$loat value a down.

Definition 4.1.16 (Arithmetic Comparisons) *If n and m are arithmetic expressions of type* `Integer`*, $\mathbb{F}$loat or* `Time` *then*

$(n \quad < \quad m), \qquad (n \quad > \quad m)$
$(n \quad <= \quad m), \qquad (n \quad >= \quad m)$

are the usual arithmetic comparison operators. The result is one of the boolean values `true` *or* `false`*. These operators compare different types, i.e.* $(3.9 <= 4T)$ *yields* `true`*, as expected.* ■

The equality and disequality predicates compare numbers in the expected way, but also every other data type.

Definition 4.1.17 (Equality and Disequality) *If n is an expression of type T and m is an expression of type Q then*

$$n \quad == \quad m \qquad \textit{and} \qquad n \quad != \quad m$$

are expressions of type `Bool`.

$n == m$ yields `true` *iff*

1. *T and Q are one of the number types* `Integer`*, $\mathbb{F}$loat and* `Time`*, and the numbers are equal, i.e. $4.0 == 4T$ yields* `true`*. ($4T$ is the long integer in the* `Time` *type.).*

2. *$T = Q$, both are enumeration types, and n and m are the same strings. This means in particular: if $T =$* `Hull`*, $Q =$* `IntvRegion`*, $n =$* `core` *and $m =$* `core` *then $n == m$ yields* `false` *(because $T \neq Q$).*

3. *$T = Q =$* `Interval` *and n and m are the same intervals (i.e. the same polygons).*

4. *$T = Q =$* `Partitioning` *and n and m are pointer-equal partitionings*

5. *$T = Q =$* `Duration` *and n and m are the same durations.*

$n\ != \ m$ yields `true` *iff $n \ == \ m$ yields* `false`. ■

4.1.2.2 Boolean Expressions

GeTS has the standard Boolean connectives: negation (-), and (‘and’ or ‘&&’), or (‘or’ or ‘||’) and exclusive or (‘xor’ or ‘^’).

Definition 4.1.18 (Boolean Expressions) *If a and b are Boolean expressions then*

$$\begin{array}{rcll} & & -a & [\texttt{Bool} \mapsto \texttt{Bool}] \\ a & \texttt{and} & b & [\texttt{Bool} * \texttt{Bool} \mapsto \texttt{Bool}] \\ a & \texttt{or} & b & [\texttt{Bool} * \texttt{Bool} \mapsto \texttt{Bool}] \\ a & \texttt{xor} & b & [\texttt{Bool} * \texttt{Bool} \mapsto \texttt{Bool}] \end{array}$$

are Boolean expressions with the corresponding meaning. ■

Flat Boolean expressions without parentheses are also allowed. The operator precedence is `xor`, `or`, `and`, i.e. `and` binds most.

4.1.2.3 Control Constructs

GeTS has the obligatory ‘if-then-else’ construct. In addition there is a `case` construct to avoid the need for nested if-then-elses. A ‘while’ loop is also available. Since GeTS is a functional language, the `while` construct needs a return value. Therefore in addition to the `while` loop body, it has a separate return expression. In the body, however, only imperative constructs (with return type `Void`) are allowed.

Definition 4.1.19 (`if-then-else`) *If c is an expression of type* `Bool` *and a and b are expressions of the same type T then*

$$\texttt{if } c \texttt{ then } a \texttt{ else } b$$

is an expression of type T.

Thus, the type of the `if` *construct is in general* $\texttt{Bool} * T * T \mapsto T$.

Exceptions are:

1. *If a is of type* $\mathbb{F}$loat*, and b of type* `Integer`*, or vice versa, then the integer is casted to* $\mathbb{F}$loat*. The type of* `if` *is in this case:*
   $\texttt{Bool} * \mathbb{F}\text{loat} * \texttt{Integer} \mapsto \mathbb{F}\text{loat}$ *or* $\texttt{Bool} * \texttt{Integer} * \mathbb{F}\text{loat} \mapsto \mathbb{F}\text{loat}$.

   Example: ‘`if true then 3 else 4.0`*’ yields 3.0 as a* $\mathbb{F}$loat *number.*

2. *If a is of type* `Time`*, and b of type* `Integer`*, or vice versa, then the integer is casted to* `Time`*. The type of* `if` *is in this case:*
   $\texttt{Bool} * \texttt{Time} * \texttt{Integer} \mapsto \texttt{Time}$ *or* $\texttt{Bool} * \texttt{Integer} * \texttt{Time} \mapsto \texttt{Time}$.

■

Notice that a mix of the type `Time` and the type $\mathbb{F}$loat is not allowed in the `if` statement.

The definition of the factorial function (4.1) is a typical example for the use of if-then-else.

Definition 4.1.20 (`case`) *If $C_1, \ldots, C_n$ are Boolean expressions and $E_1, \ldots E_n$ and D are expressions of the same type T then*

$$\texttt{case } C_1 : E_1, ..., C_n : E_n \texttt{ else } D$$

is an expression of type T. ■

The operational semantics of this `case` construct is: the conditions $C_1, \ldots, C_n$ are evaluated in this sequence. If C_i is the first condition, which yields `true` then E_i is evaluated and its result is returned as the result of `case`. If all C_i evaluate to `false` then the result of D is returned.

Exceptions for the requirement that $E_1, \ldots E_n, D$ are expressions of the same type T are: if $T = \mathbb{F}\text{loat}$ or $T =$ `Time` then some of the $E_1, \ldots E_n$ and D may have type `Integer`. These integers are automatically casted to $\mathbb{F}$loat or `Time`.
As in the 'if-then-else' construct, a mix of the types `Time` and $\mathbb{F}$loat is not allowed in the `case` body.

Definition 4.1.21 (`while`) *Let C be an expression of type* `Bool`*, $E_1, \ldots, E_n$ expressions of type* `Void` *and 'result' and expression of type T then*

$$\texttt{while}\ C\ \{E_1, ..., E_n\}\ \textit{result}$$

is an expression of type T. ■

The operational semantics of this `while` construct is: as long as the evaluation of C yields `true`, evaluate the expressions $E_1, \ldots, E_n$ in this sequence. As soon as C yields `false`, evaluate *result* and return this as value of `while`.
An iterative definition of the factorial function is a typical example where the `while` construct is used.

```
factorial(Integer n) = Let f = 1 in while(n>0){f := f*n, n := n-1} f     (4.2)
```

This example also illustrates the binding construct `Let` and the assignment operation.

Definition 4.1.22 (`Let`) *The construct*
`Let` *variable = expression*1 `in` *expression*2
of type
T
evaluates the expression1, binds the result to the variable and then evaluates expression2 under this binding.
T is the type of expression2. ■

Definition 4.1.23 (Assignment) *If x is a variable of type T, and E is an expression of type T then $x := E$ is an expression of type* `Void`. ■

This is the usual assignment operation: the result of the evaluation of E is assigned to x.

Exceptions for the requirement that x and E have the same type are: if x has type $\mathbb{F}$loat or `Time` then E may have type `Integer`. The value is automatically casted to $\mathbb{F}$loat or `Time`.
Notice that the assignment operation returns no value. It can only occur in the body of the `while` statement.

4.1.2.4 Functional Arguments

A *function call* in GeTS is an expression $\textit{name}(\textit{argument}_1, \ldots, \textit{argument}_n)$ where '*name*' is either the name of a built-in function, or the name of a previously defined function (or a function with forward declaration), or a variable with suitable functional type.

Since variables can have functional types, and GeTS allows overloading of function definitions, it needs a notation for functional arguments. A functional argument can either be just a variable with appropriate functional type, or a function name with argument type specifications, or a lambda expression. A function name with argument type specifications is necessary to choose among different overloaded functions.

Definition 4.1.24 (Functional Arguments) *A* functional argument *in GeTS is either*

1. *a variable with the appropriate functional type,*
2. *an expression name[$type_1 * \ldots * type_n$] of a previously defined function with that name and with argument types $type_1 * \ldots * type_n$, or*
3. *a lambda expression:*
   lambda($type_1$ $variable_1, \ldots, type_n$ $variable_n$) expression.
   If T is the type of 'expression' then $type_1 * \ldots * type_n \mapsto T$ is the type of the lambda-expression.
   'expression' can contain variables which are lexically bound outside the parameter list of lambda.

■

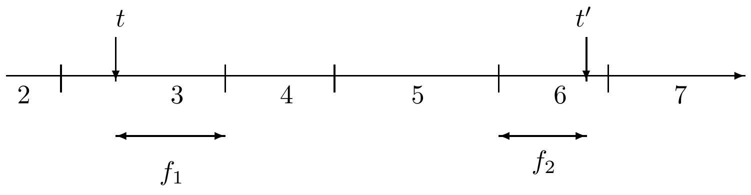
4.1.2.5 Now and Shift

Definition 4.1.25 (`now`) *The expression* `now()` *of type* `Time` *yields the current moment in time, i.e. the number of seconds from January, 1st 1970 until the time when the* `now()` *function is invoked.* ■

Notions like 'in two weeks time' or 'three years from now' etc. denote time shifts. Time shifts are basic operations for many other temporal notions. Therefore GeTS provides a `shift` function which can shift single time points as well as whole intervals by a given duration. Since in general the absolute length of durations depends on the position of the time points at the time axis, shifting time points by *durations* is no trivial operation at all.

A first specification for a `shift` function is to map a time point t to a time point t' such that $t' - t$ is just the required duration, 'two weeks' or 'three years' in the above examples. This is a *length oriented* `shift` function.

Example 4.1.26 (for Length Oriented Shift) The algorithm for this function can be best understood by the following example:



Suppose we want to shift the time point t by 3.5 partitions. First, the relative distance f_1 between t and the end of the partition containing t is measured. Suppose it is 0.75. That means from the end of the partition we need to move forward still 2.75 partitions. We can move forward 2 partitions by just adding the 2 to the coordinate 4. We end up at the start of partition 6. From there we need to move forward $f_2 = 0.75$ partitions, which is just 75% of the length of partition 6. ■

Unfortunately the length oriented shift function does not always give intuitive results. Suppose the time point t is noon at March, 15th, and we want to shift t by 1 month. March has 31 days. Therefore the distance to the end of March is exactly 0.5 months. Thus, we need to move exactly 0.5 times the length of April into April. April has 30 days. 0.5 times its length is exactly 14 days. Thus, we end up at midnight April, 14th.

This is not what one would usually expect. We would expect to shift t to the same time of the day as we started with. With the length oriented shift this happens only by chance, or when the partitions have the same length.

GeTS therefore provides also a *date oriented* shift function which avoids the above problems and gives more intuitive results. The idea is to do the calculations not on the level of reference time points, but on the level of dates. If, for example, t represents 2004/2/15, then 'in one month time' usually means 2004/3/15. That means the reference time must be turned into a date, the date must be manipulated, and then the manipulated date is turned back into a reference time. This is quite straight forward if the partitioning represents a basic time unit of a calendar system (year, month, week, day etc.), and this calendar system has a date format where the time unit occurs. In the Gregorian calendar this is the case, even for the time unit 'weeks'. 'In two weeks time' requires to turn the reference time into a date format which uses weeks. The corresponding date format uses the counting of weeks in the year (ISO 8601). For example, 2004/42/1 means Tuesday[2] in week 42 in the year 2004. In two weeks time would then be 2004/44/1.

A date oriented shift operation for partitionings which are not standard partitionings of a calendar system can usually be defined by mapping it to a date oriented shift operation for a standard partitioning. For example, if a partitioning 'semester' is defined as sequences of 6 months, one can reduce a shift in terms of semesters to a shift in terms of months. The partitioning module, which underlies the GeTS language, has for each type of partitionings a particular date oriented shift operation. For the details we must therefore refer to [33].

The next problem is to deal with fractional shifts. How can one implement, say, 'in 3.5 months time'? The idea is as follows: suppose the date format is year/month/day/hour/minute/second, and the reference time corresponds to, say, 2004/2/20/10/5/1. First we make a shift by three months and we end up at 2004/5/20/10/5/1. This is a day in May. From the date format we take the information that the next finer grained time unit is 'day'. May has 31 days. $0.5 * 31 = 15.5$. Therefore we need to shift the date first by 15 days, and we end up at 2004/5/34/10/5/1. There is still a remaining shift of half a day. The next finer grained time unit is hour. One day has 24 hours. $0.5 * 24 = 12$. Thus, the last date is shifted by 12 hours, and the final date is now 2004/5/34/22/5/1. This is turned back into a time point.

The date oriented `shift` gives more intuitive results. The drawback is that the distance between the shifted time point and the original time point need no longer be the given duration when it is measured with the `length` function (Def. 4.1.42).

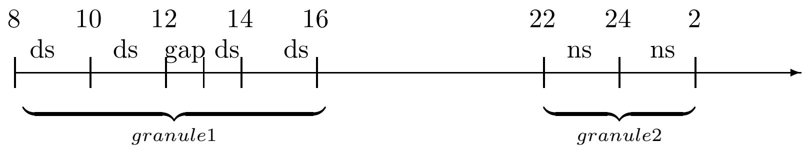
The `shift` function can not only shift time points by durations like 3.5 months. More complex durations like 3.5 months - 5 days + 3 hours are also admissible for the `shift` function.

[2]According to ISO 8601, the first day in a week is Monday. In the standard notation this is day number 1. Since we count days from 0, Monday is day 0 and Tuesday is day 1.

A shift by such a duration is executed as a sequence of shifts, first by 3.5 months, then by -5 days (backwards shift), and finally by 3 hours.

A statement like 'we must move this task by three working days' refers to a shift of time points which is measured in *granules*. GeTS offers therefore a possibility to shift time points and intervals by durations which are interpreted as *granules*. The basic idea for the algorithm which shifts a time point by a number of granules is to *turn the granules into partitions*, and to use the shift function for partitions. The method is illustrated with the following example:

Example 4.1.27 Suppose we want to model a *working day* with two shifts, a day shift from 8 am until 4 pm, with a one hour break between 12 am and 1 pm, and a night shift between 10 pm and 2 am. The labelled partitioning is `hour` with labels `ds` (for day shift) and `ns` (for night shift).



The first granule consists of 7 non-gap partitions labelled `ds`. The second granule consists of 4 non-gap partitions labelled `ns`. ■

This example shows that shifts by granules yield intuitive results only in special cases. A time point t at 9 am at day n, shifted by 2 granules, should end up at 9 am at day $n+1$. But how can we shift t by one granule, i.e. from the day shift to the night shift? GeTS provides shift operations for this and other cases as well. Whether they yield intuitive results, depends on the application.

A time point t can be: (i) within a non-gap partition of a granule, (ii) within a gap partition of a granule or (iii) within a gap partition between two granules. Suppose we want to shift t by $m = 1.5$ granules. The number of partitions to be shifted is determined as follows:

Case (i): t is within a non-gap partition p of a granule g. Suppose m is positive. Let n be the relative position of p within g (not counting internal gaps). 9 am in Example 4.1.27 is in partition 1 of a 7-partition granule (the first partition has number 0). $n = 1/7 = 0.1428$. From the start of the first granule we need to move $m + n = 1.6428$ granules forward, i.e. we need to move $n' = 0.6428$ into the second granule. Since this is a 4-partition granule, and $4 \cdot 0.6428 = 2.57$, the target partition is the *third* partition in the second granule. 9 am would be mapped to 1 am next day in the above example.

A negative shift m is treated in a similar way. The only difference is that the relative position of a partition within a granule is computed from the end of the granule and it is represented as a negative number. The relative position of the second partition within a 7-partition granule is in this case $-6/7 = -0.857$.

Case (ii): t is within a gap partition p of a granule g. Let t be at 12:30 am in Example 4.1.27. If the shift m is integer, we try to move t again into a gap partition of a granule $g + m$. If the granule $g + m$ has gaps, we determined the relative position n of the gap region containing t within g. 12:30 am is in the first gap region of a granule consisting of 2 non-gap regions. Therefore $n = 0.5$. If the target granule $g + m$ has k non-gap regions, the gap region, into which t is to be moved is the $n \cdot k$th gap region. In a second step, the relative position of the gap-partition within the gap region is mapped to a relative position of a gap partition in the target gap-area.

Negative shifts are again treated by computing relative positions as negative numbers, as in case (i).

If the target granule has no gaps, or the shift m is not integer, internal gaps are ignored and the algorithm of case (i) is applied.

Shifting $t = $ 12:30 am by $m = 2$ granules ends up at 12:30 am next day with this method. Shifting $t = $ 12:30 am by $m = 1$ granules ends up at 0:30 am next day.

Case (iii): t is within a gap partition p between two granules g_1 and g_2. If the shift m is integer then the relative position n of the gap partition p within the gap region between g_1 and g_2 is mapped to a relative position of a gap partition p' between the granules $g_1 + m$ and $g_2 + m$. If there are no gaps between $g_1 + m$ and $g_2 + m$ then the target partition p' is just the first partition of the granule $g_2 + m$.

Example 4.1.27: let t be at 5 pm and $m = 1$. 5 pm is in the second gap-partition of a 6 gap-partition region. The gap-region between the night shift and the day shift consists also of 6 gap-partitions. Therefore t is shifted to the $2/6 \cdot 6 = 2$nd gap-partition, i.e. 3 am next day.

Positive fractional shifts, for example $m = 1.5$, are treated in a relatively simple way. Let $m' \stackrel{\text{def}}{=} \lfloor m \rfloor$ be the integer part of m. The target partition p' is determined by taking the fractional part $m - m'$ as the relative position of the non-gap partitions of granule $g_2 + m'$. The exact position of t between the two granules plays no role in this case.

Shifting $t = 6 : 30$ am by 1.5 granules in the above example therefore ends up at 0:30 am next day.

Negative fractional shifts are computed by shifting the end of granule $g_1 + m'$ the fractional part $m - m'$ backwards.

Definition 4.1.28 (`shift`) *The* `shift` *function can shift a single time point by a given duration:*

$\texttt{shift}(\mathit{time}, \mathit{duration}, \mathit{asGranule}, \mathit{dateOriented})$
is of type
$\texttt{Time} * \texttt{Duration} * \texttt{Bool} * \texttt{Bool} \mapsto \texttt{Time}$.

The `shiftLength` *function determines the length of a shift:*

$\texttt{shiftLength}(\mathit{time}, \mathit{duration}, \mathit{asGranule}, \mathit{dateOriented})$
is of type
$\texttt{Time} * \texttt{Duration} * \texttt{Bool} * \texttt{Bool} \mapsto \texttt{Time}$.

■

$\texttt{shiftLength}(\mathit{time}, \mathit{duration}, \mathit{asGranule}, \mathit{dateOriented})$ is just an abbreviation for $\texttt{shift}(\mathit{time}, \mathit{duration}, \mathit{asGranule}, \mathit{dateOriented}) - \mathit{time}$.

$\texttt{shift}(\mathit{time}, \mathit{duration}, \mathit{asGranule}, \mathit{dateOriented})$ shifts the time point by the given *duration*. If *asGranule* = `true` then the partitionings in the duration are interpreted as granules, otherwise as partitions. If *dateOriented* = `true` then the shift is date oriented, otherwise it is length oriented.

4.1.2.6 Explicit Construction of Time Intervals

Fuzzy time intervals (type `Interval`) are one of the built-in data structures in GeTS. It is possible to create new empty intervals and fill them up with coordinate points. There are three ways to create new intervals in GeTS:

Definition 4.1.29 (New Time Intervals)

1. *The expression* [] *stands for the empty interval.*
2. *The expression* $[t_1, t_2]$ *of type* `Time` $*$ `Time` $\mapsto$ `Interval` *constructs a new crisp interval with boundaries* t_1 *and* t_2 *(see Example 4.1.2).*
3. *The expression* $[(t_1, y_1), (t_2, y_2)]$ *of type* `Time` $*$ $\mathbb{F}$loat $*$ `Time`$\mathbb{F}$loat $\mapsto$ `Interval` *constructs a new fuzzy interval with the given two points.*

■

Definition 4.1.30 (Extending Intervals) *The function*
`pushBack`$(I, time, value)$
of type
`Interval` $*$ `Time` $*$ $\mathbb{F}$loat $\mapsto$ `Void`
adds the point $(time, value)$ *to the end of the interval* I. I *must be an interval which was constructed with* `newInterval()` *(see Def. 4.1.29).* $time$ *must lie after the last point in the interval.* $value$ *must be a* $\mathbb{F}$loat *value between 0 and 1.* ■

The `pushBack`$(I, time, value)$ function can only ill up the interval I from the past to the future. It throws an error if $time$ is before the last time point in I.

4.1.2.7 Set Operations on Intervals

For crisp intervals there are the standard set operators: complement, intersection, union etc. These are uniquely defined. There is no choice. Unfortunately, or fortunately, because it gives you more flexibility, there are no such uniquely defined set operators for fuzzy intervals. Set operators are essentially transformations of the membership functions, and there are lots of different ones.

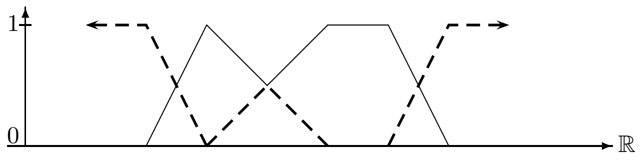
GeTS offers standard versions of the set operators, parameterized set operators of the Hamacher family, and finally set operators with transformation functions for the membership function as parameter. These allow one to customize the set operators in an arbitrary way.

Definition 4.1.31 (Complement of Intervals)
Let I *be an expression of type* `Interval`. *The complement operation for intervals comes in three versions:*

(1)	`complement`(I)	`Interval` $\mapsto$ `Interval`
(2)	`complement`(I, λ)	`Interval` $*$ $\mathbb{F}$loat $\mapsto$ `Interval`
(3)	`complement`$(I, negation\_function)$	`Interval` $*$ ($\mathbb{F}$loat $\mapsto$ $\mathbb{F}$loat) $\mapsto$ `Interval`

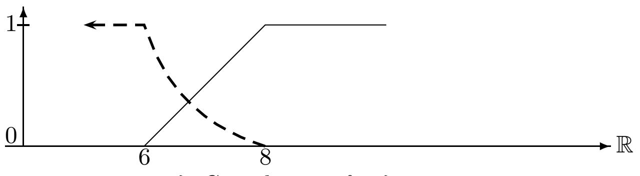
■

Version (1) is the standard complement. Each point (x, y) of the membership function of I is turned into $(x, 1 - y)$.



Standard Complement for a Fuzzy Interval

Version (2) is the *lambda-complement.* For $\lambda > -1$, each point (x, y) of the membership function of I is turned into $(x, \frac{1-y}{1+\lambda y})$. The ordinary complement is computed for $\lambda \leq -1$.



λ-Complement for $\lambda = 2$

Finally, with version (3) it is possible to submit a user defined negation function. For example, with

```
lambda_complement(Interval I, 𝔽loat lam)
= complement(I,lambda(𝔽loat y) (1-y)/(1+lam*y))
```

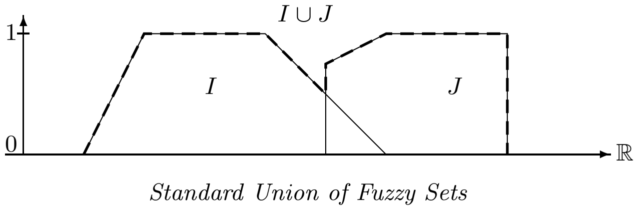
one can define the same lambda-complement with a user defined negation function.

Definition 4.1.32 (Union of Intervals) *Let I and J be expressions of type* `Interval`*. The union operation for intervals comes in three versions:*

(1)	`union`(I, J)	`Interval` $*$ `Interval` $\mapsto$ `Interval`
(2)	`union`(I, J, β)	`Interval` $*$ `Interval` $*$ 𝔽loat $\mapsto$ `Interval`
(3)	`union`(I, J, co_norm)	
	`Interval` $*$ `Interval` $*$ (𝔽loat $*$ 𝔽loat $\mapsto$ 𝔽loat) $\mapsto$ `Interval`	

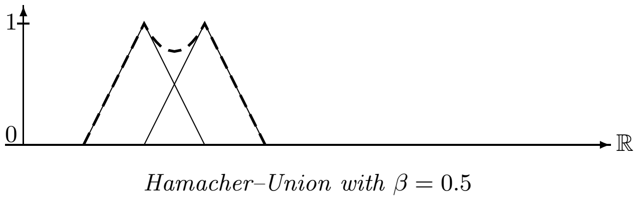
■

Version (1) is the standard union. Each pair (x, y_1) and (x, y_2) of points of the membership function of I and J is turned into $(x, max(y_1 - y_2))$.



Standard Union of Fuzzy Sets

Version (2) is the so called *Hamacher–Union.* For $\beta \geq -1$, each pair (x, y_1) and (x, y_2) of points of the membership function of I and J is turned into $(x, \frac{y_1+y_2+(\beta-1)y_1y_2}{1+\beta y_1 y_2})$. The ordinary union is computed for $\beta < -1$.



Hamacher–Union with $\beta = 0.5$

Finally, with version (3) of the union function it is possible to submit a user defined co-norm.[3] For example, with

```
Hamacher_Union(Interval I, Interval J, 𝔽loat beta)
= union(I, J, lambda(𝔽loat y1, 𝔽loat y2)
   (y1+y2+((beta - 1)*y1*y2))/(1+beta*y1*y2))
```

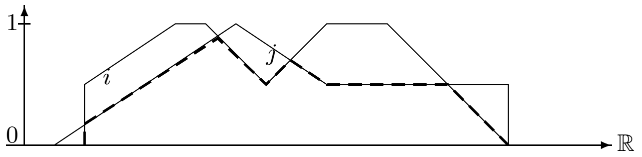
one can define the same Hamacher union with a user defined co-norm.

Definition 4.1.33 (Intersection of Intervals) *Let I and J be expressions of type* `Interval`*, The intersection operation for intervals comes also in three versions:*

(1) `intersection`(I, J) `Interval` $*$ `Interval` $\mapsto$ `Interval`
(2) `intersection`(I, J, γ) `Interval` $*$ `Interval` $*$ $\mathbb{F}$loat $\mapsto$ `Interval`
(3) `intersection`$(I, J, norm))$
`Interval` $*$ `Interval` $*$ ($\mathbb{F}$loat $*$ $\mathbb{F}$loat $\mapsto$ $\mathbb{F}$loat) $\mapsto$ `Interval`

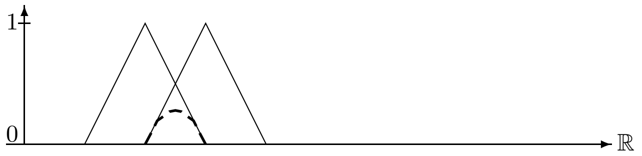
■

Version (1) is the standard intersection. Each pair (x, y_1) and (x, y_2) of points of the membership function of I and J is turned into $(x, min(y_1 - y_2))$.



Standard Intersection of Fuzzy Sets

Version (2) is the Hamacher–Intersection. For $\gamma \geq 0$, each pair (x, y_1) and (x, y_2) of points of the membership function of I and J is turned into $(x, \frac{y_1 y_2}{\gamma+(1-\gamma)(y_1+y_2-y_1 y_2)})$. The ordinary intersection is computed for $\gamma < 0$.



Hamacher–Intersection $\gamma = 0.5$

Finally, with version (3) it is possible to submit a user defined norm. For example, with

```
Hamacher_Intersection(Interval I, Interval J, Float gamma)
  = intersection(I, J, lambda(Float y1, Float y2)
           (y1*y2)/(gamma + (1-gamma)*(y1 + y2 -y1*y_2))
```

one can define the same Hamacher-Intersection with a user defined norm.

[3]Norms and co-norms are binary functions on membership values of fuzzy sets. They satisfy conditions which make sure that the corresponding set operations can be considered as union and intersection [14].

Definition 4.1.34 (Set Difference between Intervals) *Let I and J be expressions of type* `Interval`. *The set difference operation for intervals comes also in three versions:*

(1)	`setdifference`(I, J)	`Interval` $*$ `Interval` $\mapsto$ `Interval`
(2)	`setdifference`$(I, J, version)$	`Interval` $*$ `Interval` $*$ `SDVersion` $\mapsto$ `Interval`
(3)	`setdifference`$(I, J, intersection, complement)$	

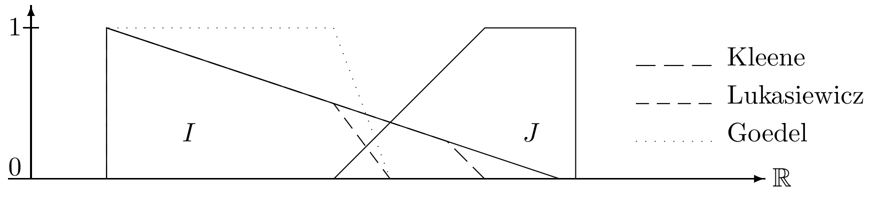
`Interval` $*$ `Interval` $*$ (`Interval` $*$ `Interval` $\mapsto$ `Interval`) $*$
(`Interval` $\mapsto$ `Interval`) $\mapsto$ `Interval`

■

(1) extends the crisp correspondence: $I \setminus J = I \cap J'$ where J' is the complement of J, `setdifference(I,J)` is therefore an abbreviation for `intersection(I,complement(J))` with standard intersection and complement functions.

(2) The second version computes the set difference operator by means of a binary function on the membership functions. The following versions are possible:

`SDVersion`	Function
`Kleene`	$(I \setminus J)(x) \overset{\text{def}}{=} min(I(x), 1 - J(x))$
`Lukasiewicz`	$(I \setminus J)(x) \overset{\text{def}}{=} max(0, I(x) - J(x))$
`Goedel`	$(I \setminus J)(x) \overset{\text{def}}{=} 0$ if $I(x) \leq J(x)$ and $1 - J(x)$ otherwise



Set Difference

(3) Finally, the third version is a generalization of the first version:

$$\texttt{setdifference}(I, J, intersection, complement) \overset{\text{def}}{=} intersection(I, complement(J))$$

intersection is a user defined binary function on intervals, and *complement* is a user defined unary function on intervals.

4.1.2.8 Predicates of Intervals

Fuzzy time intervals have many properties. They can be checked with suitable GeTS predicates.

Definition 4.1.35 (Predicates) *GeTS provides the following predicates to check the structure of an interval I:*

(1)	`isCrisp`(I)	`Interval` $\mapsto$ `Bool`
(2)	`isCrisp`$(I$, `left/right`$)$	`Interval` $*$ `Side` $\mapsto$ `Bool`
(3)	`isEmpty(I)`	`Interval` $\mapsto$ `Bool`
(4)	`isConvex(I)`	`Interval` $\mapsto$ `Bool`
(5)	`isMonotone(I)`	`Interval` $\mapsto$ `Bool`
(6)	`isInfinite(I)`	`Interval` $\mapsto$ `Bool`
(7)	`isInfinite(I,left/right)`	`Interval` $*$ `Side` $\mapsto$ `Bool`

■

`isCrisp`(I) checks whether the interval is a, possibly non-convex, crisp interval.
`isCrisp`(I, `left`) checks whether the interval is crisp at its left end. I may be infinite at this side, but the fuzzy value must be 1 in this case. Similar for `isCrisp`(I, `right`).

`isEmpty`(I) checks whether the interval is empty.

`isConvex`(I) checks whether the interval is convex. I can be non-convex even if I is crisp because it may consist of several different components (Def. 2.2.10).

`isMonotone`(I) checks whether the membership function of the interval is monotonically rising to a maximal value, and then monotonically falling again.

`isInfinite`(I) checks whether the interval is infinite.

`isInfinite`(I, `left`) checks whether the interval is infinite at the left hand side.
`isInfinite`(I, `right`) checks whether the interval is infinite at the right hand side.

The boundaries of infinite intervals are of course the infinity. Infinity has a special representation in the `Time` datatype. This can be checked with the `isInfinity` predicate:

Definition 4.1.36 (Infinity)

`isInfinity(time)`	`Time` $\mapsto$ `Bool`
`isInfinity(time,positive/negative)`	`Time` $*$ `PosNeg` $\mapsto$ `Bool`

■

`isInfinity(time)` checks whether the *time* represents an infinity.
`isInfinity(time,positive)` checks whether the *time* represents the positive infinity.
`isInfinity(time,negative)` checks whether the *time* represents the negative infinity.

The next three predicates allow one to check basic relations between time points and intervals, or between intervals and intervals.

Definition 4.1.37 (`during, isSubset, doesOverlap`)

(1)	`during`($time$, I, `core/kernel/support`)	`Time` $*$ `Interval` $*$ `IntvRegion` $\mapsto$ `Bool`
(2)	`isSubset`(I, J, `core/kernel/support`)	`Interval` $*$ `Interval` $*$ `IntvRegion` $\mapsto$ `Bool`
(3)	`doesOverlap`(I, J, `core/kernel/support`)	`Interval` $*$ `Interval` $*$ `IntvRegion` $\mapsto$ `Bool`

■

(1) `during`($time$, I, $region$) checks whether the *time* is inside the given region of the interval I.
(2) `isSubset`(I, J, $region$) checks whether the corresponding region of the interval I is a subset of the corresponding region of the interval J.
(3) `doesOverlap`(I, J, $region$) checks whether the corresponding region of the interval I overlaps the corresponding region of the interval J.

The point-interval `during` relation is one of the five point–interval relations ‘before’, ‘starts’, ‘during’, ‘finishes’ and ‘after’ for crisp intervals. Only `during` is built-in because it is one of the most frequently used relations. The other relations can easily be defined in GeTS. The point–interval `before` relation in Example 4.1.3 is such an example.

4.1.2.9 Other Features of Intervals

With the first function in this paragraph one can access the fuzzy membership value of a time point within a given fuzzy interval.

Definition 4.1.38 (`member`) *The function*

`member`$(time, I)$

of type

`Time *Interval` $\mapsto$ `Float`

returns the value of the membership function of the interval I at time point time. The value is a $\mathbb{F}$loat *number between 0 and 1.* ■

Definition 4.1.39 (Components)

1. *The function* `components`*(I) of type* `Interval` $\mapsto$ `Integer` *yields the number of components in the interval I.*

2. *The function* `component`*(I,k) of type* `Interval` $*$ `Integer` $\mapsto$ `Interval` *extracts the* $\mathtt{k}^{th}$ *component from the interval I.*

■

The function `size` below measures an interval I or parts of it by *integrating* over its membership function.

Definition 4.1.40 (`size`) *The function* `size` *comes in three versions.*

(1)	`size`(I)	`Interval` $\mapsto$ `Time`
(2)	`size`$(I,$ `core/support/kernel`$)$	`Interval` $*$ `IntvRegion` $\mapsto$ `Time`
(3)	`size`(I, t_1, t_2)	`Interval` $*$ `Time` $*$ `Time` $\mapsto$ `Time`

■

`size`(I) measures the size of the support of I.
`size`$(I,$ `core/support/kernel`$)$, measures the size of the corresponding region of I.
`size`$(I) =$ `size`$(I,$ `support`$)$.
`size`(I, t_1, t_2) measures the area of I between t_1 and t_2.

Definition 4.1.41 (sub and inf) *Let I be an interval expression.*
The function `sup`(I) *of type* `Interval` $\mapsto$ $\mathbb{F}$loat *returns the supremum of the fuzzy values of the membership function for I (usually 1).*

The function `inf`(I) *of type* `Interval` $\mapsto$ $\mathbb{F}$loat *returns the infimum of the fuzzy values of the membership function for I (usually 0).* ■

The function `length` measures the distance between two time points in terms of a partition.

Definition 4.1.42 (`length`) *The function*

`length`$(t_1, t_2, partitioning, asGranule)$

of type

`Time` $*$ `Time` $*$ `Partitioning` $*$ `Bool` $\mapsto$ $\mathbb{F}$loat

measures the distance between t_1 *and* t_2 *in terms of the given partitioning. If* $asGranule =$ `true` *then the distance is measured in terms of the length of the granules of the partitioning's labelling (without gaps).* ■

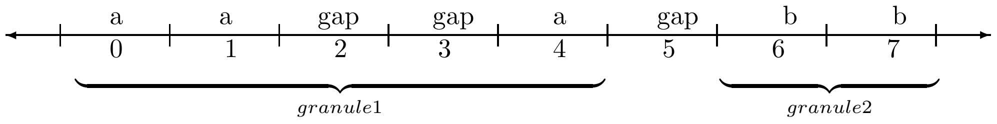
An example for determining the distance between two time points in terms of partitions is

`length(now(),shift(now(),1 day),day,false)`

is just 1.0.

The next example illustrates the `length` function in terms of granules.

Example 4.1.43 (`length` in terms of granules)
Consider a partitioning `P` with labelling a,a,gap,gap,a,gap,b,b.



The table below gives the results of `length(t1,t2,P,true)` where `t1` is the start of partition p1 and `t2` is the start of partition p2.

`t1`	`t2`	length in terms of granules
0	1	1/3
0	2	2/3
0	3	2/3
0	4	2/3
0	5	1
0	6	1
0	7	1.5
0	8	2

■

The function '`point`' below can be used to access the boundaries of the three different regions of an interval: support, core and kernel, and the first and last maximal points.

Definition 4.1.44 (`point`) *The function*
`point`$(I,$ `left/right`, `core/support/kernel/maximum`$)$
of type
`Interval` $*$ `Side` $*$ `PointRegion` $\mapsto$ `Time`
returns the position of the boundaries of I's regions:

`point`$(I,$`left,support`$)$	*yields the position of the left support boundary*
`point`$(I,$`right,support`$)$	*yields the position of the right support boundary*
`point`$(I,$`left,core`$)$	*yields the position of the left core boundary*
`point`$(I,$`right,core`$)$	*yields the position of the right core boundary*
`point`$(I,$`left,kernel`$)$	*yields the position of the left kernel boundary*
`point`$(I,$`right,kernel`$)$	*yields the position of the right kernel boundary.*
`point`$(I,$`left,maximum`$)$	*yields the leftmost position of the maximal fuzzy value.*
`point`$(I,$`right,maximum`$)$	*yields the rightmost position of the maximal fuzzy value.*

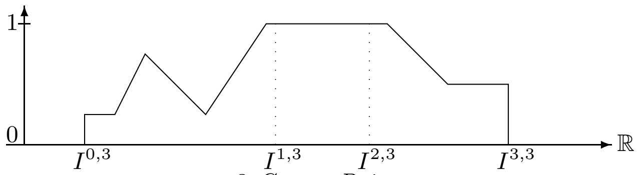
■

If `I` is just a convex crisp interval $[t_1, t_2[$ then
`point`$(I,$`left,support`$) = t_1$ and `point`$(I,$`right,support`$) = t_2$.

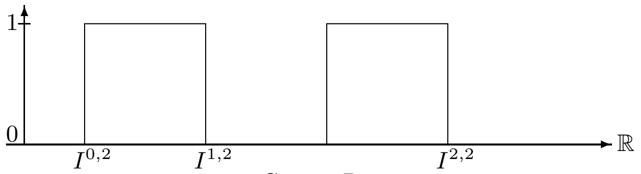
Center Points
The *n, m-center points* are used to express temporal notions like 'the first half of the year', or

'the second quarter of the year', or more exotic expressions like 'the 25th 49th of the weekend' etc. The notion of n, m-center points makes only sense for finite intervals.

Example 4.1.45 (Center Points) *The 1,2-center point $I^{1,2}$ of I splits I in two halfs of the same size (integrated over the membership function). The 1,3-center point indicates a split of I into three parts of the same size.* `centerPoint(I,1,3)` *is the boundary of the first third,* `centerPoint(I,2,3)` *is the boundary of the second third.*



$n, 3$-*Center Points*



$n, 2$-*Center Points*

■

Definition 4.1.46 (Center Points) *The function*
`centerPoint`(I, n, m)
of type
`Interval` $*$ `Integer` $*$ `Integer` $\mapsto$ `Time`
yields the (earliest) position of the n, m-center point. ■

The center points are computed such that for $n < m$:

$$\int_{centerPoint(n,m)}^{centerPoint(n+1,m)} I(x)\ dx = (\int I(x)\ dx)/m$$

4.1.2.10 Basic Manipulations of Intervals

In this paragraph we introduce some elementary transformation functions for fuzzy time intervals.

Definition 4.1.47 (Shift of Time Intervals) *The function*
`shift`(I, t)
of type
`Interval` $*$ `Time` $\mapsto$ `Interval`
shifts the interval by the given time, i.e. `shift`$(I, t)(x) = I(x - t)$ ■

Definition 4.1.48 (`cut`) *The function*
`cut`(I, t_1, t_2)
of type
`Interval` $*$ `Time` $*$ `Time` $\mapsto$ `Interval`

cuts the part of the interval I between the time points t_1 and t_2 out of I and returns it as a new interval. ∎

The `hull` function below is able to compute different hulls of a fuzzy time intervals.

Definition 4.1.49 (Hull Calculations) *The function*

`hull`$(I,$ `core/support/kernel/crisp/monotone/convex`$)$

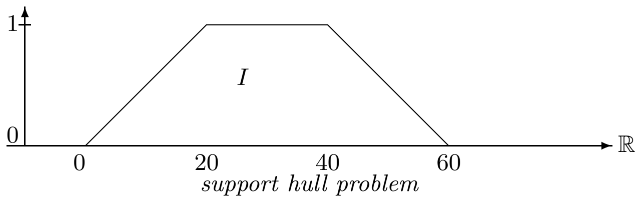
of type

`Interval` $*$ `Hull` $\mapsto$ `Interval`

computes a hull of the interval I. The second parameter determines which hull is to be computed. ∎

The `core`, `support` and `kernel` hull compute the corresponding interval regions as crisp intervals. The `core` and `support` hull may therefore consist of different components, whereas the `kernel` hull consists of at most one single component.

There is a small problem with the `support` hull. Consider the following example:



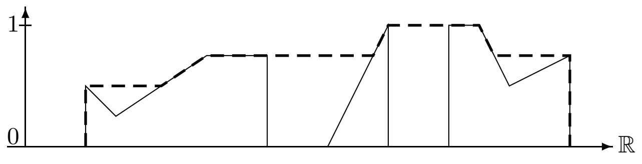
support hull problem

Since $I(0) = 0$, the support of I is the open interval $]0, 60[$. The function `hull(I,support)`, however, calculates the interval boundaries 0 and 60, which are interpreted as the half open interval $[0, 60[$. Strictly mathematical, this is not correct. In a correct implementation, however, we would have to distinguish open and half open intervals. Since the overhead for this is enormous, the current version of GeTS has to live with this error.

The `crisp` hull for crisp intervals is the usual convex hull of crisp intervals. It consists of the smallest crisp interval which contains all the components of the interval. The `crisp` hull for non-crisp intervals is the convex hull of the support of the interval. If the non-convex interval consists of one single component only, there is no difference between the crisp and support hull. In general we have

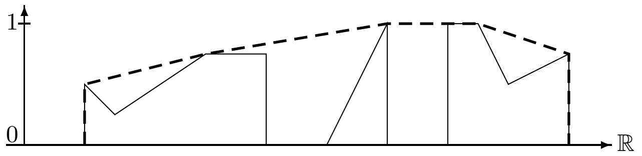
$$\texttt{hull}(I, \texttt{crisp}) = \texttt{hull(hull(}I\texttt{, support), crisp)}.$$

The `monotone` hull of an interval I is the smallest *monotone* interval which contains I. An interval is *monotone* iff its membership function rises monotonically up to a maximal point, and then falls monotonically again.



Monotone Hull of a Fuzzy Interval

The `convex` hull of an interval I is the smallest *convex* interval which contains I. The notion 'convex', which is appropriate here, is the notion of a *convex polygon*. That means, if we follow the membership function from left to right there are only right curves. The next figure illustrates this.



Convex Hull of a Fuzzy Interval

If the interval I is crisp then the crisp, monotone and convex hull are the same.

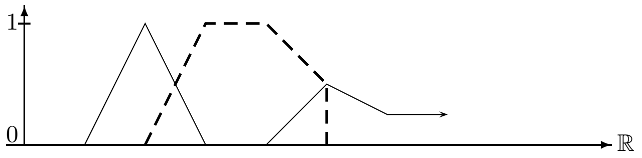
The next function can be used to extract the gaps between components of an interval. The `invert` function inverts the membership function, but only between the last maximal point of the first component and the first maximal point of the last component. `invert(I)` is zero outside these points.

Definition 4.1.50 (`invert`) *The function* `invert`(I) *of type* `Interval` $\mapsto$ `Interval` *inverts the membership function of the interval* I*:*

$$\texttt{invert}(I)(x) \overset{\text{def}}{=} \begin{cases} 1 - I(x) & \text{if } a \leq x < b \\ 0 & \text{otherwise.} \end{cases}$$

where a *is the last maximal point of the first component of* I*, and* b *is the first maximal point of the last component of* I*.* ■

Example:



Invert

`components`(`invert`(I)) yields the number of gaps in the interval I.

The `scaleup` function below multiplies the membership function of an interval I with a factor f, such that the maximal value of $I(x) * f$ is 1.

Definition 4.1.51 (`scaleup`) *The function* `scaleup`(I) *of type* `Interval` $\mapsto$ `Interval` *scales the membership function of* I *such that its maximum is 1.* ■

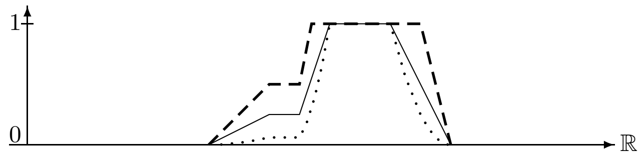
More general scaling functions are `times` and `exp`.

Definition 4.1.52 (`times` and `exp`)

$$\begin{array}{ll} \texttt{times}(I, f) & \texttt{Interval} * \mathbb{F}\text{loat} \mapsto \texttt{Interval} \\ \texttt{exp}(I, e) & \texttt{Interval} * \mathbb{F}\text{loat} \mapsto \texttt{Interval} \end{array}$$

`times`$(I, f)(x) = min(I(x) \cdot f, 1)$.
`exp`(I, e) *computes an interval such that* $exp(I, e)(x) = I(x)^e$. ■

The dashed line in the next figure indicates `times`$(I, 2)$ and the dotted line indicates `exp`$(I, 2)$.

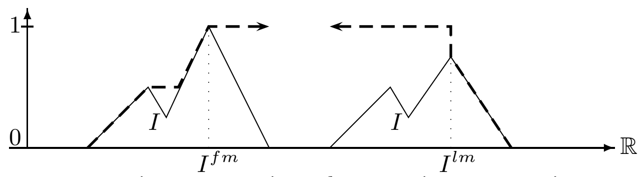


`times`$(I, 2)$ *and* `exp`$(I, 2)$

The rising part of a fuzzy time interval is crucial for a fuzzy point-interval `before` relation. The falling part, on the other hand, is crucial for a point-interval `after` relation. The rising part of an interval I can be computed by following its monotone hull up to the first maximal point, and then extending it to the infinity. Similar with the falling part.

Definition 4.1.53 (Extend to Infinity) *The function*
`extend`$(I,$`positive/negative`$)$
of type
`Interval` $*$ `PosNeg` $\mapsto$ `Interval`
extends the interval to the infinity. `extend`$(I,$`positive`$)$ *raises the membership function of the monotone hull of I to 1 after the first maximum I^{fm}.* `extend`$(I,$`negative`$)$ *raises the membership function of the monotone hull of I to 1 before the right maximum I^{lm}.* ■

Example:

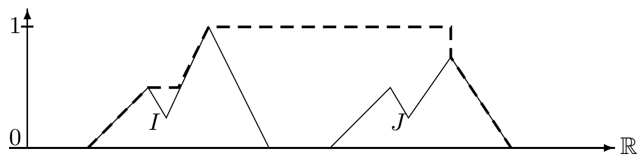


`extend(I,positive)` *and* `extend(I,negative)`

An example where the `extend` function is useful is the definition of the binary ‘until’ relation between two intervals.

```
until(Interval I, Interval J)
= intersection(extend(I,positive),extend(J,negative))          (4.3)
```

computes `until`(I, J) as the interval which lasts from the beginning of interval I until the end of interval J.



until

There is a further `extend` function in GeTS. It lengthens or shortens an interval by a certain time.

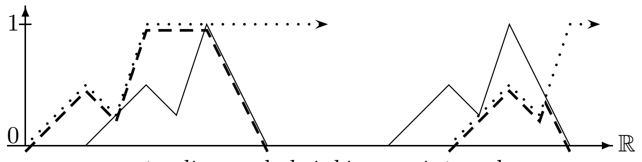
Definition 4.1.54 (Extend by a Certain Time) *The function*

`extend`$(I, length, side)$
of type
`Interval` $*$ `Time` $*$ `Side` $\mapsto$ `Interval`
extends the interval I by the given length. ■

The *side* parameter determines at which side the interval is extended. $side =$ `left` extends it at the left side, $side =$ `right` extends it at the right side. A positive *length* value causes the interval to be extended, whereas a negative *length* value causes the interval to be shrunken.

The algorithm for extending or shrinking a fuzzy interval works as follows: In a first step the interval I is split into the left/right part I_1 of the interval up to the first maximal point, and the rest I_2. I_1 is extended to the infinity. This part is shifted. If the interval is to be extended, then the union of the shifted I_1 with I_2 is computed. If the interval is to be shrinked then the intersection of the shifted I_1 with I_2 is computed. The next figure illustrates this. The dotted line shows the shifted front part of the interval. The dashed line is the result of the union/intersection.

Example:



extending and shrinking an interval
by a certain duration

The `extend` function together with `shiftLength` (Def. 4.1.28) can be used to extend an interval by a certain *duration.* For example,

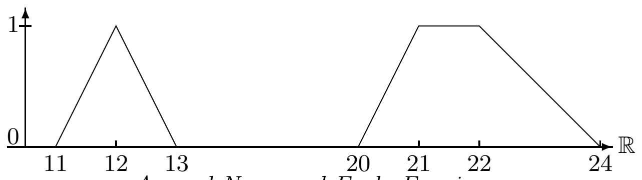
$$\texttt{extend}(I, -\texttt{shiftLength}(\texttt{point}(I, \texttt{left}, \texttt{support}), -1\ \texttt{month}, \texttt{false}, \texttt{true}), \texttt{left})$$

extends the left side of the interval I by 1 month. The month length is determined by a backwards shift of the left boundary of I's support.

Definition 4.1.55 (`integrate`) *The function*
`integrate`$(I,$ `positive/negative`$)$
of type
`Interval` $*$ `PosNeg` $\mapsto$ `Interval`
integrates the membership function of I and normalizes its value to 1. If the control parameter is `positive` *then I is integrated from left to right. If it is* `negative` *then I is integrated from right to left.* ■

An example where the `integrate` operator may be useful is the definition of *party time*

Example 4.1.56 (Birthday Party Time) Consider a database about, say, the institute's birthday parties. It may contain the entry that the birthday party for the director took place 'from around noon until early evening' of 20/7/2003. 'Around noon' is a fuzzy notion and 'early evening' is a fuzzy notion. Suppose, we have a formalization of 'around noon' and 'early evening' as the following fuzzy sets:

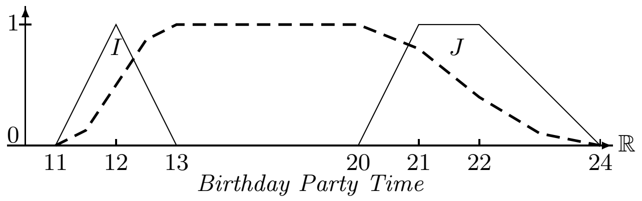


Around Noon and Early Evening

What is now the duration of the birthday party? It must obviously also be a fuzzy set. The fuzzy value of the birthday party duration at a time point x is 1 if the probability that the party started before x is 1 and the probability that the party ended after x is also 1. Therefore the fuzzy value at point x is computed by integrating over the probabilities of the start points and the end points. A natural definition would therefore be:

```
partyTime(Interval I, Interval J)
= intersection(integrate(I,positive),integrate(J,negative))
```
(4.4)

The resulting fuzzy set is:



Birthday Party Time

The dashed curve may, for example, represent the percentage of people at the party at a give time. ■

Fuzzification

Fuzzy time intervals could be defined by specifying the shape of the membership function in some way. This is in general very inconvenient. Therefore GeTS provides an alternative. The idea is to take a crisp interval and to 'fuzzify' the front and back end in a certain way. For example, one may specify 'early afternoon' by taking the interval between 1 and 6 pm and imposing, for example, a linear or a Gaussian shape increase from 1 to 2 pm, and a linear or a Gaussian shape decrease from 4 to 6 pm. Technically this means multiplying a linear or Gaussian function with the membership values.

Definition 4.1.57 (Fuzzification) *There are two different versions of the* `fuzzify` *function in GeTS. The first version allows one to specify the part of the interval I which is to be fuzzified in terms of percents of the interval length. The second version needs absolute coordinates.*

`fuzzify`(I, `linear/gaussian,left/right`, *increase*, *offset*)
of type
`Interval`, `Fuzzify`, `Side`, $\mathbb{F}$loat, $\mathbb{F}$loat $\mapsto$ `Interval`

`fuzzify`(I, `linear/gaussian,left/right`, $x1$, $x2$, *offset*)
of type
`Interval`, `Fuzzify`, `Side`, `Time`, `Time`, `Time` $\mapsto$ `Interval`

■

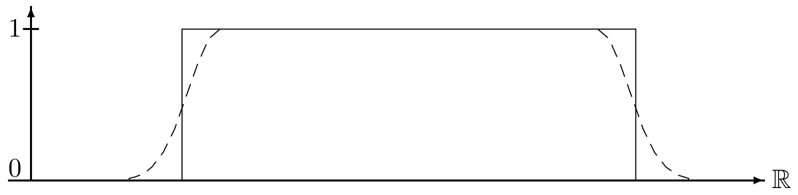
The second parameter determines whether a linear or gaussian increase is to be imposed on the interval. The third parameter determines whether the increase is from left to right or from right to left. *increase* is a Float number in percent. *increase* = 10 means that the region to be modified consists of the first/last 10% of the *kernel* of the interval. *offset* is also a float number in percent. *offset* = 20 means that the interval is to be widened by 20% of the *kernel* of the interval. To this end the fuzzified part of the interval is shifted back (second parameter = `left`) or forth (second parameter = `right`) 20% of the kernel size.

$x1$ and $x2$ in the second version of the fuzzify function allows one to determine the part of the interval to be fuzzified in absolute coordinates. `fuzzify`([0, 100], `linear`, `left`, 20, 70, 0), for example, yields a polygon [(20,0) (70,1) (100,1) (100,0)]. `fuzzify`([0, 100], `linear`, `right`, 20, 70, 0), on the other hand, yields a polygon [(0,0) (0,1) (20,1) (70,0)]. The offset widens the polygon: `fuzzify`([0, 100], `linear`, `right`, 20, 70, 20), yields [(0,0) (0,1) (60,1) (90,0)].

A function which fuzzifies both ends of an interval in the same way could be

```
f(Interval I, Float increase, Float offset)
  = intersection(extend(fuzzify(I,gaussian,left,increase,offset),positive),
                 extend(fuzzify(I,gaussian,right,increase,offset),negative))
```

$f(I, 20, 0)$ produces the following fuzzified interval.



Relative Gaussian Fuzzification

Notice that the obvious 'solution'

```
f(Interval I, Float increase, Float offset)
  = fuzzify(fuzzify(I,gaussian,right,inc,off),left,increae,offset)
```

yields no symmetric structure, because the inner `fuzzify` operation changes the kernel of the interval, such that the absolute increase and offset of the outer `fuzzify` operation are different to the absolute increase and offset of the inner `fuzzify` operation

The next example illustrate a potential use of the `fuzzify` function. We want to realize a function `beforeChristmas`. It should accept a time point t and compute a fuzzy interval, whose membership function increases for a certain time period and then stays 1.0 until Christmas. The increase is determined by two parameters, `offset` and `increase`. `offset` = 50 means that the increase should start in the middle between t and Christmas. `increase` = 20 means that the duration of the actual linear increase should be 20% of the interval length.

If t = 2004/7/1 then `beforeChristmas(t,50,10)` yields an interval whose membership function rises from 2004/9/28 until 2004/10/6/19/12 and then stays at 1.0 until 2004/12/25.

Example 4.1.58 (Before Christmas)

```
beforeChristmas(Time t, Float offset, Float increase) =
   dLet year = date(t,Gregorian_month) in
      Let christmas = time(year|12|25,Gregorian_month) in
          case (t < christmas) :
```

```
5              Let days = round(length(t,christmas,day,false),down) in
6                   fuzzify([time(year|12|25-days+round((days*offset/100)),
7                                 Gregorian_month),christmas],
8                          linear,left,increase,0),
9            (t < time(year|12|27,Gregorian_month)): []
10           else
11              Let christmas1 = time(year+1|12|25,Gregorian_month) in
12               Let days = round(length(t,christmas1,day,false),down) in
13                   fuzzify([time(year+1|12|25-days+round((days*offset/100)),
14                                 Gregorian_month),christmas1],
15                            linear,left,increase,0)
```

■

The `beforeChristmas` function considers the three cases, namely (1) that the time point t in the year y is before Christmas in this year, (2) that t is just on Christmas in this year, and (3) that t is after Christmas in this year. In case (1) the rounded number of days between t and Christmas is computed first (line 5). This number minus the offset is subtracted from `christmas` to get the left boundary of the interval to be fuzzified (line 6). The right boundary is `christmas`. The left part of the interval is fuzzified linearly with the given increase (line 6–8). If the time point t is just on Christmas (line 9) then the empty interval is returned. If t is after Christmas (case 3), then next year's Christmas is considered (line 11-15).

Integration over Pairs of Intervals
One possibility to define an interval–interval relation like '$before(I,J)$' is, to take a point–interval relation '$PIRbefore(t,J)$' and average $PIRbefore(t,J)$ over the interval I. Averaging over an interval means integrating over its membership function. For purposes like this GeTS provides two integration operations.

Definition 4.1.59 (Integration) *GeTS has the two integration functions:*

$\texttt{integrateSymmetric}(I, J, simple)$	$\texttt{Interval} * \texttt{Interval} * \texttt{Bool} \mapsto \mathbb{F}\text{loat}$ *and*
$\texttt{integrateAsymmetric}(I, J)$	$\texttt{Interval} * \texttt{Interval} \mapsto \mathbb{F}\text{loat}$

■

$\texttt{integrateAsymmetric}(I, J)$ computes $(\int I(x) \cdot J(x)\ dx)/|I|$.

$\texttt{integrateSymmetric}(I, J, simple)$ computes $(\int I(x) \cdot J(x)\ dx)/N(I, J)$

$$\text{where } N(I,J) \stackrel{\text{def}}{=} \begin{cases} min(|I|,|J|) & \text{if } simple = \texttt{true} \\ max_a(\int I(x-a)\cdot J(x)\ dx) & \text{otherwise.} \end{cases}$$

Example 4.1.4 shows an application of the asymmetric `integrate` function.

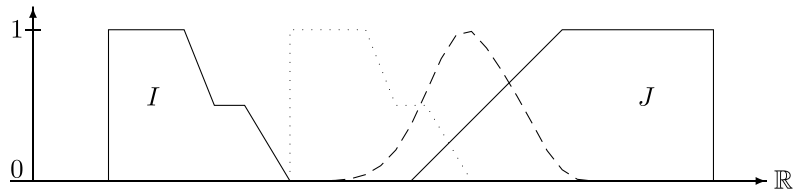
The next example shows an application of the symmetric `integrate` function. A fuzzy interval–interval relation `IIRMeets` is defined: Besides the two intervals, it takes the transformation functions F and S and integrates the interval $F(I)$ over $S(J)$. $F(I)$ should map the interval I to a finishing section of I and $S(J)$ should map the interval J to a starting section of J. The integration of $F(I)$ to $S(J)$ yields the final result.

Example 4.1.60 (Fuzzy Interval–Interval 'Meets' Relation) A possible definition for a fuzzy interval–interval meets relation is

```
IIRMeets(Interval I, Interval J, Interval->Interval F, Interval->Interval S) =
  if isEmpty(I) or isEmpty(J) or isInfinite(I,right) or isInfinite(J,left)
     then 0
  else integrateSymmetric(F(I),S(J),false)
```

■

The figure below shows the effect of the `IIRMeets` relation for suitable `F` and `S` operations. The dashed figure shows the result of `IIRMeets`$(I, J, \ldots)$ when the interval I is moved along the time axis. The dotted figure shows the position of the interval I where `IIRMeets`$(I, J, \ldots)$ is maximal.



`IIRMeets` *for Fuzzy Intervals*

GeTS contains the very special purpose function `MaximizeOverlap` which is, so far, only needed for implementing the fuzzy interval–interval overlaps relation. The classical relation I *overlaps* J has two requirements:
1. a non-empty part I_1 of I must lie before J, and
2. another non-empty part I_2 of I must lie inside J.

A generalization to fuzzy intervals encodes the first condition in the factor $1 - D(I, E^+(J))$ where D is a `during` operator. $E^+(J)$ extends the rising part of J to the infinity. Therefore $D(I, E^+(J))$ measures the part of I which is after the front part of J. $1 - D(I, E^+(J))$ then measures the part of I which is before the front part of J. This factor is multiplied with $D(I, J)$ which corresponds to the second condition. It measures to which degree I is contained in J. The product is normalized with $\max_a((1 - D(I_a, E^+(J))) \cdot D(I_a, J))$, where $I_a(x) \stackrel{\text{def}}{=} I(x - a)$. This corresponds to the maximal possible overlap when I is shifted along the time axis. This guarantees that there is a position for I where I *overlaps* $J = 1$. The normalization factor is computed with the function `MaximizeOverlap`

Definition 4.1.61 (`MaximizeOverlap`) *The function*
`MaximizeOverlap`(I, J, EJ, D)
of type
`Interval` $*$ `Interval` $*$ `Interval` $*$ (`Interval` $*$ `Interval` $\mapsto$ $\mathbb{F}$loat) $\mapsto$ $\mathbb{F}$loat
computes

$$\max_a((1 - D(\texttt{shift}(I, a), EJ)) \cdot D(\texttt{shift}(I, a), J))$$

■

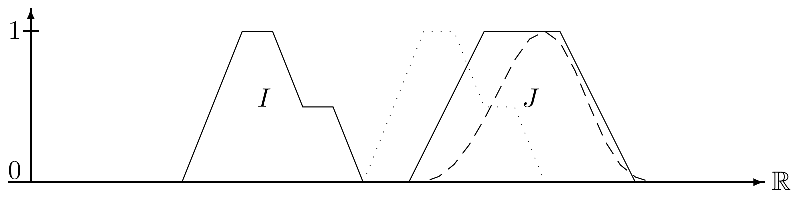
Notice that EJ can in principle be an arbitrary interval. For the encoding of the fuzzy overlaps relation, it should, however, be the extension of J to the infinity.

Example 4.1.62 *IIROverlaps*

```
IIROverlaps(Interval I, Interval J, Interval->Interval E,
            (Interval*Interval)->Float D) =
  case
    isEmpty(I) or isEmpty(J) or isInfinite(J,left) : 0,
    isInfinite(I,right) : float(point(I,left, support)  < point(J,left, support)),
    isInfinite(J,right) : float(point(I,right, support) < point(J,left, support))
  else
    Let EJ = E(J) in
       (1 - D(I,EJ))*D(I,J) / MaximizeOverlap(I,J,EJ,D)
```

■

Example 4.1.63 (IIROverlaps for Fuzzy Intervals)
This example shows the result of the `IIROverlaps` relation where the standard `IIRDuring` operator is used (with the identity function as point–interval `during` operator).



Example: Overlaps Relation

The dashed line represents the result of the overlaps relation for a time point t where the positive end of the interval I is moved to t. The dotted figure indicates the interval I moved to the position where `IIRoverlaps`(I, J) becomes maximal. ■

4.1.2.11 Date and Time

In examples 4.1.2 and 4.1.58 we have already seen applications of functions which convert time points to dates and dates to time points. The dates are sequences of integers which correspond to date formats, and these are sequences of partitionings. An example for a date format is year/month/day/hour/minute/second in the Gregorian calendar. The sequence 2004|12|3|21|43|0 in this date format is therefore the 3rd of December 2004, 9:43 pm.

The `time` function converts a date in a given date format to the corresponding time point.

Definition 4.1.64 (`time`) *The function*
`time`$(year|month|..., dateFormat)$
of type
`Integer` $* \ldots *$ `Integer` $*$ `DateFormat` $\mapsto$ `Time`
maps a date in a given date format to the time point denoted by this date. ■

The tokens *year*, *month* etc. in the `time` function are expressions of type `Integer`. There can be as many expressions as the date format has partitionings. For example, the year/month/day/hour/minute/second date format in the Gregorian calendar has 6 partitionings. Therefore there are in this case at most 6 `Integer` expressions allowed in the `time` function.

Examples:
`time(2004,Gregorian_month)` = 1072915231 (1st of January 2004)

`time(2004|1+1,Gregorian_month)` = 1075593631 (1st of February 2004)
`time(2004|2|2,Gregorian_week)` = 1073347231 (6th of January 2004)
`Gregorian_week` is the date format year/week/day/hour/minute/second. Therefore 2004|2|2 is the second day in the second week in the year 2004.

The `dLet` construct in the next definition is a kind of inverse to the `time` function. It computes for a given time point and a date format a date representation as a sequence of `Integer`s and binds the variables to these `Integer`s in a similar way as the `Let` construct.

Definition 4.1.65 (`dLet`) *The expression*

$$\texttt{dLet}\ year|month|... = \texttt{date}(time, dateFormat)\ \texttt{in}\ expression$$

binds the variables $year, month, \ldots$ to the integers which correspond to the date denoted by 'time', in the given date format.
'expression' is then evaluated under this binding.
The type of `date` *is* `Time*DateFormat` $\mapsto$ `Integer`^{n} *where $n \leq$ maximal number of partitionings in the date format.*

■

Example:
'`dLet` $y|m|d|h =$ `date`$(0,$`Gregorian_month`$)$ `in` $y+m+d$' yields 1973 because the time point 0 corresponds to the first of January 1970. Therefore $y = 1970$, $m = 1$, $d = 1$ and $h = 0$.

4.1.2.12 Partitionings and Labels

GeTS has a number of functions for reckoning with time points, partitions and labels. The `partition` function was already introduced in Example 4.1.1.

Definition 4.1.66 (`partition`) *The* `partition` *function maps time points to intervals, which represent partitions.*

(1) `partition`$(time, partitioning)$
  `Time` $*$ `Partitioning` $\mapsto$ `Interval`
(2) `partition`$(time, partitioning, n, m)$
  `Time` $*$ `Partitioning` $*$ `Integer` $*$ `Integer` $\mapsto$ `Interval`

■

The first version computes the interval which corresponds to the partition containing *time*. The second version computes an interval $[t_1, t_2[$ as follows: If i is the coordinate of the partition containing *time* then t_1 is the start of partition $i+n$ and t_2 is the end of the partition $i+m$.

If instead of the partition as interval, only the boundaries are needed, one can use the `partitionBoundary` function.

Definition 4.1.67 (partition boundary) *The function*
`partitionBoundary`$(time, partitioning,$ `left/right`$)$
of type
`Time` $*$ `Partitioning` $*$ `Side` $\mapsto$ `Time`
computes the left/right boundary of the partition containing time. ■

Although partitionings are in general infinite mathematical structures, their validity region may be limited. GeTS has two functions for getting information about the boundaries of the valid regions of a partitioning.

Definition 4.1.68 (Valid Regions of Partitionings) *The function*
`partitioningIsBounded`(P, `left/right`)
of type
`Partitioning` $*$ `Side` $\mapsto$ `Bool`
checks whether the valid region of the partitioning P is bounded at the given side.

The function
`partitioningBoundary`(P, `left/right`)
of type
`Partitioning` $*$ `Side` $\mapsto$ `Time`
returns the boundary of the partitioning at the given side. If there is no bound at this side then a representation of infinity is returned. ■

The next function is `which`. It can, for example, be used to compute which week in the year is now, or which day in the semester is now.

Definition 4.1.69 (`which`) *The function*
`which`($time, P, Q, inclusion, asGranule$)
is of type
`Time` $*$ `Partitioning` $*$ `Partitioning` $*$ `Inclusion` $*$ `Bool` $\mapsto$ `Time`.
■

The function is explained for the two *asGranule* case:

Case $asGranule =$ `false`**:**
Consider the following example:
`which(now(),week,year,bigger_part_inside,false)`.
It computes, which week of the year is now.

The `which` function first computes the starting point t_Q of the Q-partition containing *time*. In the example, it would be the beginning of the current year. Then it determines the P-partition for t_Q. In the example, it is the first week in the year. What counts as the 'first' P-partition p depends on the parameter *inclusion*:

inclusion $=$ `subset`**:** p is the leftmost P-partition after t_Q.

inclusion $=$ `overlaps`**:** p is the leftmost P-partition containing t_Q.

inclusion $=$ `bigger_part_inside`**:** p is the leftmost P-partition whose bigger part comes after t_Q. (This is suitable for counting weeks within a year).

If n is the coordinate of p and m is the coordinate of the P-partition containing *time* then $m - n$ is returned by the `which` function. If `now()` is first of January then the call
`which`(`now()`, `week`, `year`, `bigger_part_inside`, `false`)
returns 0 as the number of the first week in the year.

Notice that the result of the `which` function is of type `Time`. The `Time` datatype is in this case just to be taken as a potentially very big integer, and not as a time point.

Case $asGranule =$ **`true`:**
The partitionings P and Q are in this case interpreted as granules. t_Q is the start of the Q-granule containing *time*. If *time* is between two granules then t_Q is the end of the Q-granule before t_Q.

The first P granule p depends again on the parameter *inclusion*:

inclusion = **`subset`:** p is the leftmost P-granule after t_Q.

inclusion = **`overlaps`:** p is the leftmost P-granule containing t_Q. If t_Q is between two P-granules then p is the leftmost P-granule after t_Q.

inclusion = **`bigger_part_inside`:** p is the leftmost P-granule whose bigger part comes after t_Q.

A special case is that *time* lies before the start of p. In this case the `which` function returns the value -1 to indicate that the counting is not possible.

In the normal case the `which` function counts from granule p as number 0 the P-granules until it reaches *time*. If *time* is between two P-granules then the counting stops before *time*. The value of the counter is returned.

The functions in the next definition deal with labels of partitions. Notice that labels are not just strings. They are special data structures, such that, for example, two labels with the same name are identical.

Definition 4.1.70 (Basic Functions for Labels) *The function*
`label`(*time*, *partitioning*)
of type
`Time` $*$ `Partitioning` $\mapsto$ `Label`
returns the label of the partition containing time. If there is no labelling defined, it returns a NULL label.

The function
`isLabel`(*label*)
of type
`Label` $\mapsto$ `Bool`
checks whether the label is not the NULL label.

The function
`isGap`(*label*)
of type
`Label` $\mapsto$ `Bool`
checks whether the label is the gap label.

The function
`LabelName`(*name*)
of type
`String` $\mapsto$ `Label`
turns a string (without quotes) into a `Label`*.* ■

The `extractLabelled` function below can be used to extract from an interval all partitions with a given label, for example all Tuesdays of a labelled day partitioning.

Definition 4.1.71 (`extractLabelled`) *The function*

`extractLabelled`$(I, label, partitioning, inclusion, intersect)$

of type

`Interval` $*$ `Label` $*$ `Partitioning` $*$ `SplitInclusion` $*$ `Bool` $\mapsto$ `Interval`

extracts partitions in the interval I with the given label. ■

The `extractLabelled` function maps through all partitions of the given partitioning which are labelled with the given label, and which overlap with the interval $[a, b[$ where a is the left boundary of the interval and b is the right boundary of the interval. An error is thrown if a or b are the infinity.

For each such partition p a condition is tested which depends on the parameter *inclusion*.

inclusion = **`align`:** the condition is always true.

inclusion = **`subset`:** p must be a subset of I's support.

inclusion = **`overlaps`:** p must overlap with I's support.

inclusion = **`bigger_part_inside`:** the bigger part of p must be a subset of I's support.

If the parameter *intersect* = `false` then all partitions p which meet the condition are joined into the resulting (crisp) interval.

If the parameter *intersect* = `true` then the intersection of I with all partitions p which meet the condition are joined into the resulting interval. The result may now be a fuzzy interval.

The function below is for constructing intervals which represent granules.

Definition 4.1.72 (`nextGranule`) *The function*

`nextGranule`$(time, partitioning, label, n, withGaps)$

of type

`Time` $*$ `Partitioning` $*$ `Label` $*$ `Integer` $*$ `Bool` $\mapsto$ `Interval`

constructs a new interval which represents a granule. ■

The interval is constructed as follows:
Case 1: *time* is inside a granule with the given *label*.

If $n = 0$ then this granule is computed. Otherwise the n^{th} next/previous (if $n < 0$) granule with this label is computed.
Case 2: *time* lies outside a granule with the given *label*.

If $n = 0$ then the empty interval is returned. Otherwise the n^{th} next/previous (if $n < 0$) granule with this label is computed.

Finally an interval is constructed and returned which represents the granule. If *withGaps* = `true` then this interval may be non-convex to exclude the gap partitions within a granule.

4.1.2.13 Control Constructs for Operations on Intervals

GeTS has two basic control constructs for operations on parts of intervals. The `componentwise` control construct allows one to apply an operation to each component of an interval and to combine the results of each application with a combination function.

Definition 4.1.73 (`componentwise`) *The following function applies an operation to each component of an interval and combines the results with a combination operator. It comes in two versions, without and with an end test.*

$\texttt{componentwise}(I, \mathit{initialObject}, \mathit{operation}, \mathit{combination})$
of type
$[\texttt{Interval} * T * (\texttt{Interval} \mapsto T) * (T * T \mapsto T) \mapsto T]$

$\texttt{componentwise}(I, \mathit{initialObject}, \mathit{operation}, \mathit{combination}, \mathit{endTest})$
of type
$[\texttt{Interval} * T * (\texttt{Interval} \mapsto T) * (T * T \mapsto T) * (T \mapsto \texttt{Bool}) \mapsto T]$

■

I is the interval whose components are considered.
operation is the operation which is applied to the components of I. It generates results of type T (which is determined by the type of *initialObject*).
combination is the operation which is used to combine the results of the application of *operation*.
initialObject is the object which is returned when I is empty, and which is used to combine it with the very first result of *operation*. Typically, $\mathit{initialObject} = []$ ($T = \texttt{Interval}$) or $\mathit{initialObject} = 0T$ ($T = \texttt{Time}$).
endTest is a predicate which is applied to the intermediate results. The loop is terminated and the intermediate result is returned as soon as *endTest* yields `true`.

Examples:
$\texttt{componentwise}(I, [], \texttt{lambda}(\texttt{Interval}\ J)\ \texttt{hull}(J, \texttt{crisp}),$
$\qquad\qquad \texttt{lambda}(\texttt{Interval}\ K, \texttt{Interval}\ L)\ \texttt{union}(K, L))$

computes the crisp hull for each component of the interval I separately and then joins them into one single crisp, possibly non-convex, interval.

$\texttt{componentwise}(I, 0.0,$
$\quad \texttt{lambda}(\texttt{Interval}\ J)\ \texttt{length}(\texttt{point}(J, \texttt{left}, \texttt{support}),$
$\qquad\qquad \texttt{point}(J, \texttt{right}, \texttt{support}), \texttt{month}, \texttt{false})$
$\quad \texttt{lambda}(\mathbb{F}\text{loat}\ n, \mathbb{F}\text{loat}\ m)\ n + m)$

computes the lengths of the support of (a finite) interval I in terms of months.

split:
The `split` function below is in principle similar to the `componentwise` function. The difference is that the interval is not taken apart into its components, but it is split into subintervals of a given length. A function is applied to these split parts, and a combination function combines the partial results into a final result.

Definition 4.1.74 (`split`) *The* `split` *function also comes in two versions, without and with an end test.*

$\texttt{split}(I, duration, asGranule, dateOriented, initialObject, operation,$
$combination, region, forward, inclusion, sequencing, intersect)$
is of type
$\texttt{Interval} * \texttt{Duration} * \texttt{Bool} * \texttt{Bool} * T * (\texttt{Interval} \mapsto T) * (\texttt{Interval} * \texttt{Interval} \mapsto T)*$
$\texttt{IntvRegion} * \texttt{Bool} * \texttt{SplitInclusion} * \texttt{Sequencing} * \texttt{Bool} \mapsto T$

$\texttt{split}(I, duration, asGranule, dateOriented, initialObject, operation,$
$combination, region, forward, inclusion, sequencing, intersect, endTest)$
is of type
$\texttt{Interval} * \texttt{Duration} * \texttt{Bool} * \texttt{Bool} * T * (\texttt{Interval} \mapsto T) * (\texttt{Interval} * \texttt{Interval} \mapsto T)*$
$\texttt{IntvRegion} * \texttt{Bool} * \texttt{SplitInclusion} * \texttt{Sequencing} * \texttt{Bool} * (T \mapsto \texttt{Bool}) \mapsto T.$

■

In order to explain the `split` function in detail, we must introduce some auxiliary functions. They are not part of GeTS, but used internally.

The *startpoint* and *endpoint* functions compute the starting point for the split.

Definition 4.1.75 (*startpoint* and *endpoint*) The function $startpoint(t, P, inclusion, asGranule)$ of type $\texttt{Time} * \texttt{Partitioning} * \texttt{SplitInclusion} * \texttt{Bool} \mapsto \texttt{Time}$ computes the starting point of a forward split as follows:

Case $asGranule = \texttt{false}$**:**

- **Case** $inclusion = \texttt{align}$**:** return t
- **Case** $inclusion = \texttt{subset}$**:** return the starting point s of the leftmost P-partition such that $t \leq s$.
- **Case** $inclusion = \texttt{overlaps}$**:** return the starting point of the leftmost P-partition containing t.
- **Case** $inclusion = \texttt{bigger\_part\_inside}$**:** return the starting point of the leftmost P-partition p such that the bigger part of p comes after t.

Case $asGranule = \texttt{true}$**:**

- **Case** $inclusion = \texttt{align}$**:** If t is between two granules g_1 and g_2 then return the starting point of g_2, otherwise return t.
- **Case** $inclusion = \texttt{subset}$**:** return the starting point of the leftmost granule after t.
- **Case** $inclusion = \texttt{overlaps}$**:** If t is between two granules g_1 and g_2 then return the starting point of g_2, otherwise return the starting point of the leftmost granule containing t.
- **Case** $inclusion = \texttt{bigger\_part\_inside}$**:** If t is between two granules g_1 and g_2 then return the starting point of g_2, otherwise return the starting point of the leftmost granule whose bigger part comes after t. Gaps within a granule are not measured.

A similar function $endpoint(t, P, inclusion, asGranule)$ computes the starting point of a backwards split:

Case $asGranule = \texttt{false}$**:**

Case *inclusion* = `align`**:** return t

Case *inclusion* = `subset`**:** return the end point s of the rightmost P-partition such that $s \leq t$.

Case *inclusion* = `overlaps`**:** return the endpoint of the rightmost P-partition containing t.

Case *inclusion* = `bigger_part_inside`**:** return the endpoint of the rightmost P-partition p such that the bigger part of p comes before t.

Case *asGranule* = `true`**:**

Case *inclusion* = `align`**:** If t is between two granules then return the endpoint of the rightmost granule before t, otherwise return t.

Case *inclusion* = `subset`**:** return the endpoint of the rightmost granule before t.

Case *inclusion* = `overlaps`**:** If t is between two granules then return the endpoint of the rightmost granule before t, otherwise return the endpoint of the rightmost granule containing t.

Case *inclusion* = `bigger_part_inside`**:** If t is between two granules then return the endpoint of the rightmost granule before t, otherwise return the endpoint of the rightmost granule whose bigger part comes before t. Gaps within a granule are not measured.

■

The functions *advance* and *retract* below compute the start of the next split part.

Definition 4.1.76 (*advance* **and** *retract*) The function $advance(t, P, sequencing, asGranule)$ of type `Time` ∗ `Partitioning` ∗ `Sequencing` ∗ `Bool` ↦ `Time` computes for the end time t of a (forward) split part the start time of the next split part:

Case *asGranule* = `false`**:**

Case *sequencing* = `sequential`**:** return t;

Case *sequencing* = `overlapping`**:** return the start of the P-partition containing t;

Case *sequencing* = `with_gaps`**:** if t is the start of the P-partition containing t then return t, otherwise return the start of the following P-partition.

Case *asGranule* = `true`**:**

Case *sequencing* = `sequential`**:** if t is between two granules then return the start of the next granule, otherwise return t.

Case *sequencing* = `overlapping`**:** if t is between two granules then return the start of the next granule, otherwise return the start of the granule containing t.

Case *sequencing* = `with_gaps`**:** if t is between two granules then return the start of the next granule. If t is the start of a granule then return t, otherwise return the start of the granule which follows the granule containing t.

The corresponding function $retract(t, P, sequencing, asGranule)$ computes for the start time t of a (backward) split part the end time of the next split part:

Case $asGranule =$ `false`**:**

- **Case** $sequencing =$ `sequential`**:** return t;
- **Case** $sequencing =$ `overlapping`**:** return the end of the P-partition containing t;
- **Case** $sequencing =$ `with_gaps`**:** if t is the end of the P-partition containing t then return t, otherwise return the end of the previous P-partition.

Case $asGranule =$ `true`**:**

- **Case** $sequencing =$ `sequential`**:** if t is between two granules g_1 and g_2 then return the end of g_1, otherwise return t.
- **Case** $sequencing =$ `overlapping`**:** if t is between two granules g_1 and g_2 then return the end of g_1, otherwise return the end of the granule containing t.
- **Case** $sequencing =$ `with_gaps`**:** if t is between two granules g_1 and g_2 then return the end of g_1. If t is the end of a granule then return t, otherwise return the end of the granule which is before the granule containing t.

■

Back to the function
`split`$(I, duration, asGranule, dateOriented, initialObject, operation,$
$combination, region, forward, inclusion, sequencing, intersect, endTest)$.

The parameters I, *initialObject*, *operation* and *combination* have the same meaning as for the `componentwise` function (Def. 4.1.73).

The interval I can be split in forward direction ($forward =$ `true`) or in backward direction ($forward =$ `false`).

Region to be split:
The parameter *region* (= `core`, `support` or `kernel`) determines the region $[a, b[$ of the interval I which is to be split. a is the leftmost point of the region and b is the rightmost point of the region. An error is thrown if a or b is the infinity.

The split loop:
Let P_0 be the first partitioning which occurs in *duration*. Let $A = initialObject$ be the accumulator for the operation.

Case $forward =$ `true`**:**

Let $t_0 = startpoint(a, P_0, inclusion, asGranule)$ be the starting point for the split. The `split` command performs the following (forward) loop:

```
$while(t_0 < b)\{$
  let $t_1 := shift(t_0, duration, asGranule, dateOriented)$;   (Def. 4.1.28)
  let $J := [t_0, t_1[$  be the split part;
  $if\ intersect =$ true $J := I \cap J$;
  $A := union(A, operation(J))$;
  $if(endTest(A) =$ true$)\ return\ A$;
  $t_0 = advance(t_1, P, sequencing, asGranule); \}$
$return\ A$
```

Case $forward = \texttt{false}$**:**

Let $t_1 = endpoint(b, P_0, inclusion, asGranule)$ be the starting point for the split.

The `split` command now performs the following (backwards) loop:

```
$while(t_1 > a)\{$
  let $t_0 := shift(t_1, neg(duration), asGranule, dateOriented)$;    (Def. 4.1.28)
  let $J := [t_0, t_1[$ be the split part;
  $if\ intersect = \texttt{true}\ J := I \cap J$;
  $A := union(A, operation(J))$;
  $if(endTest(A) = \texttt{true})\ return\ A$;
  $t_0 = retract(t_1, P_0, sequencing, asGranule); \}$
$return\ A$
```

4.2 Summary

The GeTS language is a special purpose functional specification and programming language for temporal notions. It has a basic set of general purpose functional and imperative programming language features. In addition there are a number of built-in data structures and functions which are specific for this application. The most important ones are time points, fuzzy temporal intervals and labelled partitionings of the time line.

GeTS is not a stand alone programming language. It must be part of a host system which provides these data structures and which invokes the GeTS application programming interface.

The GeTS constructs were carefully chosen as a compromise between simplicity and easy usage. Future applications will show whether this goal has been achieved.

Chapter 5

Relations Between Fuzzy Time Intervals

Abstract: This paper serves two purposes. The first purpose is to introduce a new approach for defining point–interval and interval–interval relations for fuzzy time intervals. The basic idea for the interval–interval relations is to extend corresponding point–interval relations to interval–interval relations by averaging (integrating) the point–interval relation over the second interval. The new approach is compared with an existing approach by Nagypál and Motik.

The second purpose is to show how these relations can be easily defined with the GeTS–language (GeTS stands for GeoTemporal Specifications). This is a typed functional language with a large number of built–in data types and functions for manipulating temporal notions. The definitions of the point–interval and interval–interval relations are therefore given directly in the GeTS–language.

5.1 Motivation and Introduction

Time points and time intervals are the basic concepts in many formalisations of temporal notions. In order to make basic concepts useful for practical applications, one has to define operations on them and relations between them. If we take, for example, two time points t_1 and t_2 and a linear ordering of the time structure, which we assume throughout this paper, there are the basic relations $t_1 < t_2$, $t_1 = t_2$ and $t_1 > t_2$. The relations between a time point t and a (crisp) time interval I are slightly more complex. First of all, one needs to distinguish whether I is convex or not. If it is not convex it consists of several unconnected subintervals. Secondly, one can consider or ignore the underlying metric of the time axis. If the metric is ignored and I is convex there are the usual five point–interval relations:

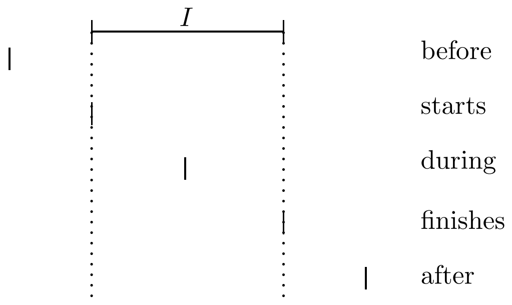


Figure 5.1: Point–Interval Relations

These relations are well defined even if I is infinite, or if one distinguishes whether I is closed or open at one or both sides. If I is open at the left side then t *starts* I may be true even if $t\,during\,I$ is false. For a non-convex interval I there are some more relations: 't is within the n'the component of I', 't is in a gap of I', 't is within the n'th gap of I' etc.

The metric of the time axis gives rise to more relations. With a metric one can measure the location and length of the interval. Therefore, at least for finite intervals I, it makes sense to define 't is in the first half of I', or more general, 't is in the n'th m'th of I'. These relations can also be obtained in an indirect way by cutting out the first half, or, more general, the n'th m'th of I, and evaluating point–interval relations between t and the cut out part of I. This way one can easily define relations like 't is before (after, ...) the n'th m'th of I'.

In the next stage we can consider interval–interval relations. Allen's interval–interval relations [1] are the basic relations between two crisp intervals (without metric).

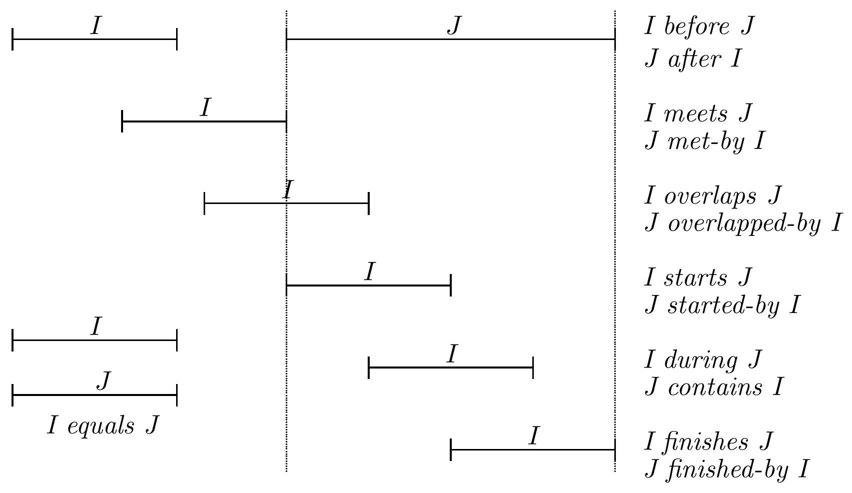


Figure 5.2: Interval–Interval Relations

Natural language has a lot more interval–interval relations They, however, usually rely on the time metric. 'is long before' or 'is close to' are examples.

Things get a lot more complicated if we consider fuzzy time intervals instead of crisp time intervals. Fuzzy time intervals can be used to represent fuzzy notions like 'around noon', 'late night', 'during sunrise' etc. If the intervals are in fact fuzzy, one expects that the relations 'before', 'during' etc. are also no longer simple relations with a Boolean result, but binary functions with a fuzzy value as result. Unfortunately, there are no unique and natural ways to generalise the relations between crisp intervals to relations between fuzzy intervals. There are many different possibilities, and it depends on the applications, which one is the most appropriate one. A tool for representing and manipulating fuzzy intervals should therefore provide several of these possibilities, or, even better, allow the user to define his favourite version of the relations.

The purpose of this paper is therefore twofold:

1. a new approach for interval–interval relations between fuzzy time intervals is presented. The idea of this new approach is to take a suitable point–interval relation and extend it to an interval–interval relation by averaging the point–interval relation over the second interval. The details of this approach are presented and compared to other approaches;

2. the fuzzy interval–interval relations are used as a case study for a part of the CTTN system (Computational Treatment of Temporal Notions) [28]. The CTTN system provides data structures and operations for various basic temporal notions. In particular it contains time points, crisp and fuzzy time intervals [31], labelled partitionings for representing periodic temporal notions and calendar systems [33]. In addition it has the GeTS (GeoTemporal Specification) language for specifying and computing with application specific temporal notions. In this paper it is shown how the GeTS–language can be used to define crisp and fuzzy point–interval and interval–interval relations.

No argument is given in this paper that a particular version of the relations is good for a particular application. Instead, we want to give evidence that the CTTN system and in particular the GeTS–language is good for specifying even very complex relations in a simple and intuitive way.

5.2 Point–Interval Relations

The five basic relations between a time point and a *crisp* interval I are *before*, *starts*, *during*, *finishes* and *after* (cf. Fig. 5.1). If the intervals possess a metric, which is the case for time intervals over the real numbers, there are infinitely many more point–interval relations. Examples are ‘during the first half’ or ‘in the middle of the third quarter’.

If the metric is given in terms of time units of a calendar system there are even more complex point–interval relations. ‘before the year 2006’, for example, can mean ‘some weeks or months before 2006’, but not ‘1000 years before 2006’. Linguists analyse the precise meaning of such expressions. The GeTS–language provides constructs for defining these more complex relations, but this is not subject of this paper.

For non-convex intervals there are even more point–interval relations, for example ‘between the components’ or ‘between the second and third component’ etc.

A point–interval relation R can also be represented as a function which maps an interval to an interval. For example, the ‘before’ relation can be represented as the function which maps an interval I to the interval J containing all the points before I. This formalisation of point–interval relations as functions of type `Interval` $\mapsto$ `Interval` can now be generalised to fuzzy intervals in a way that fuzzy point–interval relations yield fuzzy values instead of Boolean values.

Definition 5.2.1 (Point–Interval Relations as Functions)

1. *A fuzzy point–interval relation $R(t, I)$ is a function that maps a time point t and an interval I to a fuzzy value.*
2. *If I is a fuzzy interval and R' is a function of type* `Interval` $\mapsto$ `Interval` *then R with the definition:*
$$R(t, I) \stackrel{\text{def}}{=} R'(I)(t)$$
is the corresponding fuzzy point–interval relation. ■

With this definition one can turn any `Interval` $\mapsto$ `Interval` function into a fuzzy point–interval relation. Since there are infinitely many of them, it is by no means obvious how to characterise some of them as, for example, fuzzy ‘before’ relations, or fuzzy ‘starts’ relations etc. One could, for example, require that an `Interval` $\mapsto$ `Interval` function B represents a fuzzy ‘before’ relation if in the limit case where I is a crisp interval, $B(I)$ is the ordinary crisp ‘before’ function. This, however, is a property which is not always desired. For example, it should be possible to assign a non-zero fuzzy value to the statement ‘the movie ends *before* the concert’, even if the movie ends one minute after the concert starts.

In fact, there is no clear mathematical characterisation of `Interval` $\mapsto$ `Interval` functions which allows one to distinguish the corresponding different kinds of point–interval relations. Now, what can we do? A pragmatic approach is to develop tools which allows the user to specify his favourite version of fuzzy point–interval relations in an easy and intuitive way. In order to make the tool more useful, it should contain some predefined versions of the different point–interval relations, which are good enough for most applications.

Three Versions of the Point–Interval Relations

For each of the standard point–interval relations, *before*, *starts*, *during*, *finishes* and *after*, we provide three different versions. All three versions are functional, i.e. they return an interval as a result.

Version 1:
The first version is essentially a pure crisp version of the relation. The type of the function is `Interval` * `Region` $\mapsto$ `Interval`. `Region` is one of the key words `support`, `core`, `kernel`, `maximum`. This version takes the corresponding crisp region $S(I)$, $C(I)$, $K(I)$ or $M(I)$ of the interval I (Def. 2.2.4) and computes the crisp relation.

Version 2:
This is the most general version. It has some extra parameters which are functions for manipulating the interval in a certain way.

Version 3:
Version 3 is a specialisation of version 2. The extra parameters are instantiated with suitable concrete functions. This version is the 'standard' fuzzy point–interval relation.

5.2.1 Point–Interval 'Before' and 'After' Relations

The *before* function maps an interval I to an interval containing all the points before I.

Definition 5.2.2 (Point–Interval 'Before' Functions)

```
1. PIR::Before(Interval I, Region R) =
        if isEmpty(I) then []
        else Let t=point(I,left,R) in
                 if(isInfinity(t)) then [] else [(t,1.0),(t,0.0)]

2. PIR::Before(Interval I, Interval->Interval C, Interval->Interval E) =
        if isEmpty(I) then []
        else C(E(I))

3. PIR::Before(Interval I) =
        if isEmpty(I) then []
        else complement(extend(I,positive))
```

■

Version 1:
This version turns the interval into a crisp interval containing the points before the corresponding region of the interval.

The expression `Let t= point(I,left,R)` causes that this value is bound to the local variable `t`. `[(t,1.0),(t,0.0)]` constructs a crisp interval $]-\infty, t[$.

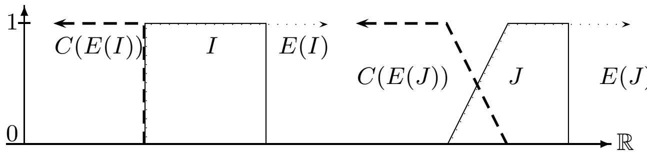
Version 2:
This is the most general version of a 'before' function. Both extra parameters `C` and `E` are in principle arbitrary (total) `Interval` $\mapsto$ `Interval` functions. The idea behind this function is that the parameter `E` is a function which isolates the front part of the interval and extends it to $+\infty$. The parameter `C` must be a complement function which complements the front part.

Version 3:
is an instance of version 2 where `E` is the function `lambda(I) extend(I,positive)` and `C` is the standard complement function: `complement`$(I)(t) = 1 - I(t)$ (Sect. 2.2.4). `PIR::Before(Interval I)` is actually a shortcut for

```
PIR::Before(I,lambda(Interval J) complement(J),
              lambda(Interval J) extend(J,positive))
```

We illustrate version 3 and version 2 with a few examples.

Example 5.2.3 (`PIR::Before`) *$E(I) \overset{\text{def}}{=}$* `extend(`$I$`,positive)` *and $C(I) \overset{\text{def}}{=}$* `complement(`$I$`)`.



`PIR::Before`

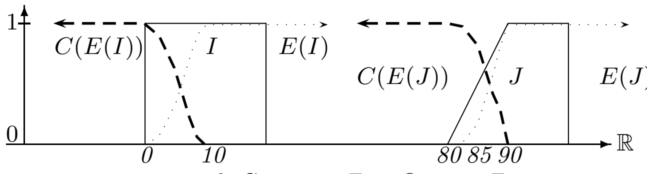
■

The left picture shows `PIR::Before(I)` where `I` is a crisp interval. This yields the ordinary crisp 'before' relation. `PIR::Before(I)` yields the same result as version 1, `PIR::Before(I,R)` where `R` is any of the admissible key words.

The next examples show how the *before*–relation can be made fuzzy, even for crisp intervals. The pictures are produced with the following call of the `PIR::Before` relation.

```
PIR::Before(I,lambda(J) complement(J),
              lambda(J) extend(fuzzify(I,gaussian,left,50,0),positive))
```

Example 5.2.4 (Gaussian `PIR::Before`**)** *$E(I)\overset{\text{def}}{=}$*`extend(fuzzify(`$I$`,gaussian,left,50,0),positive)` *and $C(I) \overset{\text{def}}{=}$* `complement(`$I$`)`.



`PIR::Before` *with Gaussian Fuzzification Function*

■

The expression `fuzzify(`I`,gaussian,left,50,0)` (Def. 2.2.33) causes that the front 50% of I is multiplied with a gaussian function. `extend(fuzzify(`I`,gaussian,left,50,0),positive)` then yields the front part of the fuzzified I, extended to the infinity.

Notice that `fuzzify(extend(`I`,positive),gaussian,left,50,0)` does not work. The reason is that `extend(`I`,positive)` produces an infinite interval, and 50% of an infinite interval is not defined.

'After':
The point–interval 'after' functions are very analogous to the corresponding 'before' functions. In particular, the general version 2 has exactly the same definition as version 2 of PIR::Before. In order to get an acceptable result of version 2 of PIR::After, one has to pass a function like

```
lambda(Interval J) extend(J,negative)
```

as the `E` parameter to PIR::After. `extend(`J`,negative)` extracts the back part of J and extends it to $-\infty$.

Definition 5.2.5 (Point–Interval 'After' Functions)

```
1. PIR::After(Interval I, Region R) =
        if isEmpty(I) then []
        else Let t = point(I,right,R) in
                 if(isInfinity(t)) then [] else [(t,0.0),(t,1.0)]

2. PIR::After(Interval I, Interval->Interval C, Interval->Interval E) =
        if isEmpty(I) then []
        else C(E(I))

3. PIR::After(Interval I) =
        if isEmpty(I) then []
        else complement(extend(I,negative))
```

■

5.2.2 Point–Interval 'Starts' and 'Finishes' Relations

The standard *starts*-relation t *starts* I for crisp intervals I yields 1 if t is the starting point of I, and 0 everywhere else. This corresponds to an almost empty fuzzy set with a single peak right at the start of I. The generalisation of this simple definition to fuzzy intervals is version 1 in Def. 5.2.6 below. `t = point(I,left,R)`, where `R` is again one of the key words `support`, `core`, `kernel` or `maximum`, determines the left boundary of the corresponding region as the start of the interval. The generated interval is `[t,t+1]`, which internally is mapped to the half open interval $[t, t+1[$. Since it is half open, its fuzzy value is 1 for the time point t and 0 for the time point $t+1$, as expected.

Definition 5.2.6 (Starts)

```
1.  PIR::Starts(Interval I, Region R) =
        if isEmpty(I) then []
        else Let t=point(I,left,R) in
                 if(isInfinity(t)) then [] else [t,t+1]

2. PIR::Starts(Interval I, Interval->Interval E, Interval->Interval B,
        (Interval*Interval)->Interval Intersect) =
        case isEmpty(I) or isInfinite(I,left): [],
             isCrisp(I,left): scaleup(Intersect(E(I),shift(B(I),1)))
        else                  scaleup(Intersect(E(I),B(I)))

3. PIR::Starts(Interval I) =
        case isEmpty(I) or isInfinite(I,left): [],
             isCrisp(I,left): Let t=point(I,left,support) in [t,t+1]
        else scaleup(intersection(extend(I,positive),PIR::Before(I)))
```

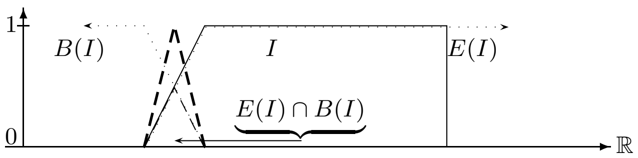
■

Version 1:
This version yields a single peak even if the front part of the interval rises smoothly.

Version 2:
The more general version 2 is more fuzzy in this case. Version 2 of PIR::Starts accepts two extra `Interval -> Interval` functions as arguments. The first one, `E`, is supposed to extract the front part of the interval and extend it to $+\infty$. The second one, `B`, should be one of the point–interval 'before' functions. `scaleup(Intersect(E(`I`),B(`I`)))` intersects `E(`I`)` and `B(`I`)` and scales it up such that the maximum fuzzy value of the intersection is 1.

Version 3:
This version is an instance of version 2 with `E = lambda(Interval `J`) extend(`J`,positive)` and
`B = lambda(Interval `J`) PIR::Before(`J`)`. `Intersect` is the standard intersection function (Sect. 2.2.4).

Example 5.2.7 (Standard `PIR::Starts`-function) *Version 3 of* `PIR::Starts` *generates the following result. The dashed line indicates the scaled up intersection.*



`PIR::Starts`

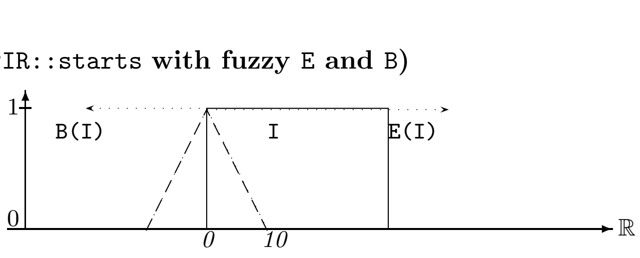
■

There is a small technical problem if we apply `Intersect`$(E(I), B(I))$ to a crisp interval. With the choice of the `E` and `B` functions as in Example 5.2.7, the intersection of `E(I)` and `B(I)` is empty. The second clause in version 2 of `PIR::Before` takes care of this by shifting `B(I)` one time unit into the future. This way, however, one gets again just a single peak at the start of `I`. If this peak is to be widened in order to get a more fuzzy 'starts' relation even for crisp intervals, there are two possibilities. Either the definition of `PIR::Before` is changed for this case, or you take other `B` and `E` functions. The following definitions

```
E = lambda(Interval J) extend(fuzzify(J,left,linear,30,0),positive) and
B = lambda(Interval J) complement(extend(fuzzify(J,left,linear,30,10),positive)),
```

for example, work for finite intervals. The next picture shows how they fuzzify the start of the interval.

Example 5.2.8 (`PIR::starts` with fuzzy `E` and `B`)



`PIR::starts`

■

'Finishes':
The definition of the functions `PIR::Finishes` below are analogous to the definition of the `PIR::Starts` functions. The difference is that the `E` function is expected to extract the end of the interval and extend it to $-\infty$. Instead of the 'before' function B, we need an 'after' function A.

Definition 5.2.9 (Finishes)

```
1. PIR::Finishes(Interval I, Region R) =
        if isEmpty(I) then []
        else Let t=point(I,right,R) in
                 if(isInfinity(t)) then [] else [t,t+1]

2. PIR::Finishes(Interval I, Interval->Interval E, Interval->Interval A,
        (Interval*Interval)->Interval Intersect) =
        case isEmpty(I) or isInfinite(I,right):  [],
             isCrisp(I,right): scaleup(Intersect(shift(E(I),1),A((I))))
        else                   scaleup(Intersect(E(I),A(I)))

3. PIR::Finishes(Interval I) =
        case isEmpty(I) or isInfinite(I,right):  [],
             isCrisp(I,right):  Let t=point(I,right,support) in [t,t+1]
        else scaleup(intersection(extend(I,negative),PIR::After(I)))
```

■

5.2.3 Point–Interval 'During' Relations

The simplest version of the 'during' relation is just the identity: t *during* $I \overset{\text{def}}{=} I(t)$. This yields 0 for all t outside I and the fuzzy membership value for all t inside I. This is version 4 of `PIR::During` below.

Definition 5.2.10 (`PIR::During`)

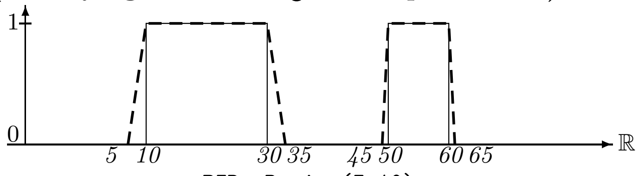
```
1. PIR::During(Interval I, Region R) =  hull(I,R)

2. PIR::During(Interval I, Interval->Interval O,
              (Interval*Interval)->Interval Union) =
        if isEmpty(I) then  []
        else componentwise(I,[],O,Union)

3. PIR::During(Interval I, Float percent) =
        if isEmpty(I) then  []
        else componentwise(I,[],
              lambda(Interval J) fuzzify(J,linear,percent,percent),
              lambda(Interval J, Interval K) union(J,K))

4. PIR::During(Interval I) =  I
```

■

Version 1:
If we insist that `PIR::During` yields a crisp result, we can take version 1. `hull`(I, R) computes $S(I)$, $C(I)$, $K(I)$ or $M(I)$ (Def. 2.2.4), depending on the parameter R.

Versions 2 and 3:
Versions 2 and 3 of `PIR::During` are more fuzzy versions of *during*. They can return non-zero values even for crisp intervals and time points outside the interval. The basic idea of version 2 is to apply a fuzzification operator O to all components of the interval and to join the fuzzified components with the `Union` operator. Version 3 makes this explicit. It uses the fuzzification operator `fuzzify(`J`,linear,percent,percent)` which fuzzifies the (finite) interval at both ends. The amount of fuzzification is controlled by the `percent` parameter. The next picture shows an example.

Example 5.2.11 (Fuzzifying `PIR::During` for crisp intervals)



`PIR::During(I,10)`

■

5.2.4 Point–Interval Relations for Non-Convex Intervals

Non-convex crisp intervals consist of several unconnected components. In this case a time point t can also be in any of the gaps between the components. This gives rise to a general 'in the gap' point–interval relation. More specific is a 'in the k'th gap' relation, which determines the particular gap containing t. The same relations can also be defined for non-convex fuzzy intervals. Fuzzy intervals, however, can be non-convex in two ways. They can consist of several components, i.e. their membership function drops down to 0 and rises again. This is a similar kind of non-convexness as for crisp intervals. They can, however, also be non-convex if they consist of a single component, but the membership function drops down and then rises again. The definition of `PIR::InTheGap` below yields nontrivial results only if the interval consists of several components.

Definition 5.2.12 (In The Gap)

```
1. PIR::InTheGap(Interval I) =
          if isEmpty(I) then []
          else invert(I)

2. PIR::InTheGap(Interval I, Integer k) =
          if isEmpty(I) then []
          else component(invert(I),k)
```
■

The `invert` function in the definitions above inverts the gap regions of the intervals (see Sec. 2.2.30).

5.2.5 Point–Interval Relations for Intervals with Metric

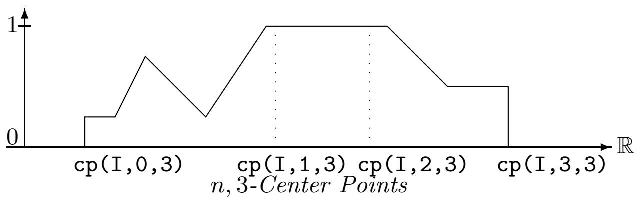
If the underlying time structure has a metric, it is possible to measure the length of the intervals and to subdivide them into halfs or thirds etc. This gives rise to relations like ‘in the first half’ or ‘in the second third’, or, in general, ‘in the n’th m’th. The function `PIR::During` below cuts out the n’th m’th of the given interval.

Once a relation like ‘in the first half’ is defined, it is very natural to define the relation ‘in the middle of the first half’. The crisp definition 2 in Def. 5.2.13 below of this relation has just one single peak in the middle of the first half. The version 3 in Def. 5.2.13 of `PIR::InTheMiddle` widens this peak by cutting out a slice around this peak. The width of this slice is controlled by the parameter `k`. The larger the `k` the smaller the slice.

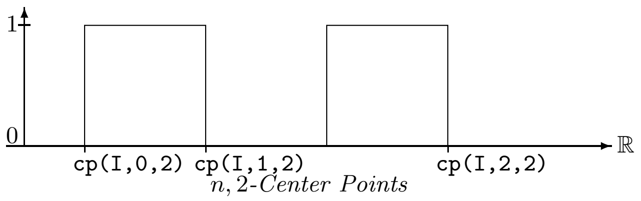
Definition 5.2.13 (Relations with Metric)

```
1. PIR::During(Interval I, Integer n, Integer m) =
        if(isEmpty(I) or (n < 0) or (m < 0) or (m <= n))  then []
        else cut(I,centerPoint(I,n,m),centerPoint(I,n+1,m))

2. PIR::InTheMiddle(Interval I, Integer n, Integer m) =
        Let t = centerPoint(I,2n+1,2m) in [t,t+1]

3. PIR::InTheMiddle(Interval I, Integer n, Integer m, Integer k) =
        if(isEmpty(I) or (n < 0) or (m < 0) or (m <= n) or (k < 0))  then []
        else Let k1 = pow(2,k) in
             Let n1 = 2*n + 1  in
             Let m1 = k1*2*m   in
                 cut(I,centerPoint(I,k1*n1 - 1,m1),centerPoint(I,k1*n1 + 1,m1))
```
■

Crisp intervals can easily be measured by summing up the length of the components. Fuzzy intervals can be measured by integrating over their membership functions. The function `centerPoint(I,n,m)` in the definitions above computes the `n,m` center point of the interval. For example, `centerPoint(I,1,2)` computes the middle point of the interval. If t = `centerPoint(I,1,2)` then the integral over `I` up to t has the same value as the integral over `I` after t.



n, 3-*Center Points*



n, 2-*Center Points*

Middle Points:
The middle point between the center points `centerPoint(I,n,m)` and `centerPoint(n+1,m)` is just `centerPoint(I,2n+1,2m)`. For example, the middle point in the first half of `I` is `centerPoint(I,1,4)` and the middle point in the second half is `centerPoint(I,3,4)`. This is exploited in the definition of `PIR::InTheMiddle`. The parameter `k` controls the size of the slice around this middle point which is cut out of `I`.

5.3 Interval–Interval Relations

Allen's seven interval relations [1] are the basic relations between two crisp intervals (cf. Fig. 5.2). The extension of these relations to fuzzy intervals is neither obvious nor unique. There is one approach proposed by Nagypál and Motik [25], and this paper introduces a second approach.

5.3.1 Nagypál and Motik's Interval–Interval Relations

The basic idea of Nagypál and Motik's approach is to extend the requirements for the crisp interval–interval relations, which are mostly conditions on the end points of the intervals, to conditions on the starting and finishing sections of the fuzzy intervals. We summarise how these relations work. More details are given in the original paper [25].

Definition 5.3.1 (Nagypál and Motik's Interval–Interval Relations)
Let I and J be two fuzzy intervals.

$$
\begin{array}{lcl}
\text{before}(I,J) & \overset{\text{def}}{=} & sup_t(((1-E^-(I)) \cap (1-E^+(J)))(t)) \\
\text{meets}(I,J) & \overset{\text{def}}{=} & min(inf_t((E^-(I) \cup E^+(J))(t)), \\
 & & \quad inf_t(((1-E^-(I)) \cup (1-E^+(J)))(t))) \\
\text{overlaps}(I,J) & \overset{\text{def}}{=} & min(sup_t((E^+(I) \cap (1-E^+(J)))(t)), \\
 & & \quad sup_t((E^-(I) \cap E^+(J))(t)), \\
 & & \quad sup_t(((1-E^-(I)) \cap E^-(J))(t))) \\
\text{starts}(I,J) & \overset{\text{def}}{=} & min(inf_t(((1-E^+(I)) \cup E^+(J))(t)), \\
 & & \quad inf_t((E^+(I) \cup (1-E^+(J))(t))), \\
 & & \quad sup_t(((1-E^-(I)) \cap E^-(J))(t))) \\
\text{during}(I,J) & \overset{\text{def}}{=} & min(sup_t(((1-E^+(I)) \cap E^+(J))(t)), \\
 & & \quad sup_t(((1-E^-(I)) \cap E^-(J))(t))) \\
\text{finishes}(I,J) & \overset{\text{def}}{=} & min(inf_t((E^-(I) \cup (1-E^-(J))(t))), \\
 & & \quad inf_t(((1-E^-(I)) \cup E^-(J))(t)), \\
 & & \quad sup_t((E^+(I) \cap (1-E^+(J)))(t))) \\
\text{equals}(I,J) & \overset{\text{def}}{=} & min(inf_t((E^-(I) \cup (1-E^-(J))(t))), \\
 & & \quad inf_t(((1-E^-(I)) \cup E^-(J))(t)), \\
 & & \quad inf_t(((1-E^+(I)) \cup E^+(J)))(t)) \\
 & & \quad inf_t((E^+(I) \cup (1-E^+(J)))(t)).
\end{array}
$$

$E^+(I)$ *stands for* `extend(`I`,positive)` *and* $E^-(I)$ *stands for* `extend(`I`,negative)`. ■

The relations give quite intuitive results for finite fuzzy intervals consisting of one component only, and without much internal structure. The definitions work also for infinite intervals, but the results may not be very intuitive. Nagypál and Motik's relations behave on crisp intervals just like Allen's interval–interval relations. This is different to the approach which is presented in this paper.

5.3.2 Operator Based Interval–Interval Relations

The main requirements for the 'operator based' version of the fuzzy interval–interval relations are:

1. even for two crisp intervals we want a fuzzy value as result. The result should of course be 1 if the classical relation yields true, but it should not necessarily jump to 0 when the classical relation yields false;

2. the relations should work for fuzzy time intervals regardless if they consist of one or more components or if they are finite or infinite.

The basic idea for the operator versions of the relations is very simple: since we have the point–interval relations, we can extend the point to an interval in the relation by integrating over the interval's membership function. Thus, the operator version of the interval–interval relations are essentially averaged point–interval relations. The functional version of a point–interval relation is the *operator* which is used for the averaging process.

In the rest of this section we present the GeTS definitions of four different versions of the interval–interval relations. Version 1 is the crisp version, where the corresponding crisp region (support, core, kernel or maximum) is the basis for the computation. Version 2 is the most general version where some extra functional parameters are used to compute the relation. Version 3 is an instance of version 2 where default values are used for the extra parameters.

Finally, version 4 is Nagypál and Motik's version. Version 1 has a Boolean value as result, whereas all the others have a fuzzy value as result.

Visualisation of the Interval–Interval Relations
A fuzzy interval–interval relation between two concrete fuzzy intervals I and J yields a single fuzzy value as a result. In order to show the effect of a particular definition for such a relation we move the interval I in discrete steps along the time axis and compute for each shift of I the relation to J. The resulting fuzzy values are collected in a new fuzzy interval. The GeTS function `showRelation` below shifts the interval I and collects the points (t, y) in a new interval K. t is the time coordinate of the right end of the kernel of the shifted I and $y = F(I, J)$ is the result of the fuzzy relation F. The figure in Example 5.3.3 shows the result of `showRelation`. There are many more examples of this kind in the subsequent sections. The right end of the kernel of the shifted interval I is, however, not always used for the time coordinate of the generated interval. In some examples we use the middle of the kernel.

```
showRelation(Interval I, Interval J, (Interval*Interval)->Float F,
             Time distance, Integer steps) =
        Let K = [] in
            while (steps >= 0) {
                  pushBack(K,point(I,right,kernel),F(I,J)),
                  I := shift(I,distance),
                  steps := steps - 1}
            K;
```

5.3.3 Interval–Interval 'before' Relations

Four different versions of the 'before' relation are presented as GeTS definitions.

Definition 5.3.2 (Interval–Interval 'before' Relations)

```
1. IIR::Before(Interval I, Interval J, Region R) =
        case isEmpty(I): false,
             isEmpty(J): false
        else (point(I,right,R) <= point(J,left,R))

2. IIR::Before(Interval I, Interval J, Interval->Interval B) =
        case isEmpty(I) or isEmpty(J): 0.0,
             isInfinite(I,right) or isInfinite(J,left): 0.0,
             (point(I,right,support) <= point(J,left,support)): 1.0,
             isInfinite(I,left):
                 Let K = intersection(I,J) in
                     if(isEmpty(K)) then 1.0 else integrateAsymmetric(K,B(J))
        else integrateAsymmetric(I,B(J))

3. IIR::Before(Interval I, Interval J) = IIR::Before(I,J,PIR::Before[Interval])

4. IIR::NMBefore(Interval I, Interval J) =
        case isEmpty(I) or isEmpty(J): 0.0,
             isInfinite(I,right) or isInfinite(J,left): 0.0
        else sup(intersection(complement(extend(I,negative)),
                              complement(extend(J,positive))))
```

■

Version 1

This is Allen's version of the crisp *before*–relation. The parameter `R` (`support`, `core`, `kernel` or `maximum`) determines the crisp part of the intervals which are to be compared.

The `<=` in the expression `point(I,right,R) <= point(J,left,R)` may need an explanation. All intervals are interpreted as half open intervals with integer boundaries. If, for example, $I = [0, 10[$ and $J = [10, 20[$ then 10 belongs to J and not to I. Therefore all points of I, including the 10, are definitely before J. Therefore `<=` is appropriate here.

Version 2 and 3

Version 2 and version 3, as a special case of version 2, are the operator based *before*–relations. The first three cases in version 2 of `IIR::Before` concern trivial cases where the result is obvious. The last two cases are the really interesting ones.

Version 2 for Finite Intervals
The definition for finite intervals I and arbitrary intervals J is:

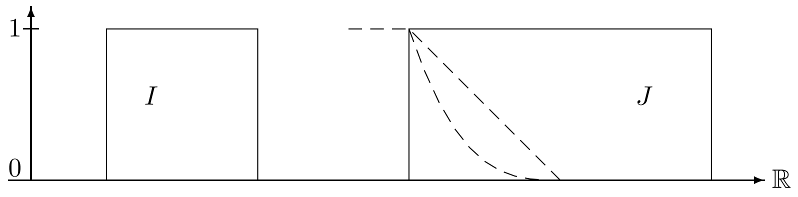
$$\textit{before}(I, J) \stackrel{\text{def}}{=} \int I(t) \cdot B(J)(t)\ dt/|I|$$

where B is a point–interval *before*-relation. The idea of this definition is to average the point–interval *before*-relation over the interval I. The normalisation factor is just $|I|$. The rationale behind this is the following: if J is finite at the left side then $B(J)(t) = 1$ if t is small enough. If we move the interval I into the area where $B(J)(t) = 1$ for all t in this area then the integral becomes $\int I(t) \cdot B(J)(t)\ dt = \int I(t) \cdot 1\ dt = |I|$, such that $\textit{before}(I, J) = 1$. Thus, $|I|$ as normalisation factor yields the right result.

The built-in function `integrateAsymmetric`$(I, B(J))$ in GeTS computes the normalised integral $\int I(t) \cdot B(J)(t)\ dt/|I|$.

The next picture illustrates version 3 for two finite crisp intervals. B is the standard point–interval *before* operator of Example 5.2.3. For this particular operator B and crisp intervals I and J, `IIR::Before`(I, J) yields the percentage of points in I which are before J (in the usual crisp sense).

Example 5.3.3 (`IIR::Before` for crisp intervals) *The upper dashed line in the picture indicates the fuzzy value at position t if the end of interval I is at this position. The fuzzy value has dropped to 0 only if the interval I is moved completely into J. A steeper decrease can be enforced if the result of* `IIR::Before`(I, J) *is, for example, exponentiated with an exponent* > 1. *(GeTShas an* `exp` *function for this purpose.) The lower dashed line in the figure therefore indicates* `IIR::Before`$(I, J)^3$.



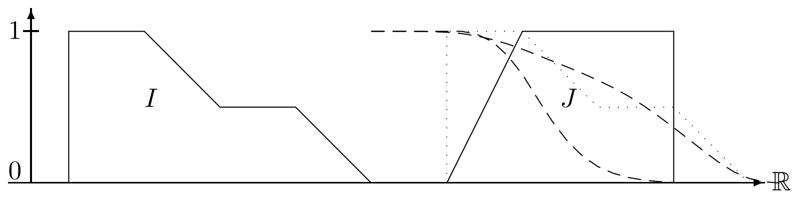
`IIR::Before`(I,J) *and* `IIR::Before`$(I,J)^3$

■

One may argue whether it is desirable to have a 'before' relation where the standard parameters force the fuzzy value down to 0 only when the interval I is completely contained in J. The counter argument is that a smooth decrease can reveal more information about the structure of I and J than a steep drop. A steep drop to 0 can always be achieved by exponentiating the result with a large enough exponent.

The next picture shows the `IIR::Before`–relation for real fuzzy intervals.

Example 5.3.4 (`IIR::Before` for fuzzy intervals) *The upper dashed line in the picture indicates the result of the* `IIR::Before`*–relation at position t if the positive end of the interval I is moved to t. The lower dashed line is the upper dashed line exponentiated with the exponent 10. The dotted line represents the position of the interval I when the result value is dropped to 0.*



`IIR::Before`(I,J) *and* `IIR::Before`$(I,J)^{10}$

■

Version 2 for Infinite Intervals

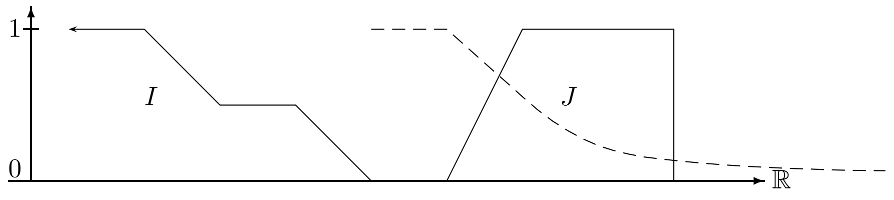
If the interval I is positive infinite, then nothing can be after I. Therefore the relation `IIR::Before`(I,J) must yield 0. If the interval J is negative infinite, then nothing can be before J. Therefore the relation `IIR::Before`(I,J) must also yield 0.

If I is negative infinite and J is finite or positive infinite then `IIR::Before`(I,J) may well be not false. The problem is how to measure the degree of 'beforeness' in this case. Since I is infinite, $\int I(t)\cdot B(J)(t)\ dt$ will always be infinite. An alternative is to take instead of I only the intersection between I and J and to measure the degree of 'beforeness' of $I\cap J$. Since J is not negative infinite, $I\cap J$ is finite. The definition is then

$$\mathit{before}(I,J) \overset{\text{def}}{=} \int (I\cap J)(t)\cdot B(J)\ dt/|I\cap J|.$$

This integral is again computed with the GeTS function `integrateAsymmetric`$(I\cap J, B(J))$

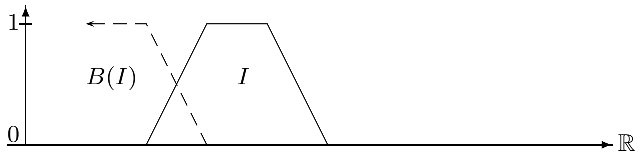
Example 5.3.5 (`IIR::Before` for infinite intervals) *This picture shows the development of* `IIR::Before`(I, J) *when I is negative infinite and its positive end is moved along the time axis. The fuzzy value drops down, but not to 0. It remains constant after a while.*



before *for Infinite Intervals*

■

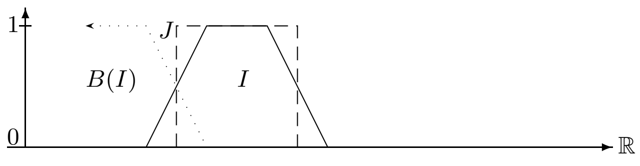
Properties of the `IIR::Before`–Relation for Finite Intervals:
The crisp relation on crisp sets is irreflexive: $\neg$*before*(I, I) holds for all I. `IIR::Before`$(I, J) = 0$ holds also for crisp sets. `IIR::Before`$(I, I) = 0$ means in this case that 0% of I is before I itself. This does not longer hold if I is fuzzy. The following picture illustrates the phenomenon:



Counterexample for Irreflexivity

The intersection of $B(I)$ with I is to a certain degree before I. Therefore `IIR::Before`$(I, I) > 0$.

The crisp *before* relation is asymmetric: $\forall I, J$ *before*$(I, J) \Rightarrow \neg$*before*(J, I). A similar property also holds for the `IIR::Before`–relation on crisp sets: if a non-zero fraction of I is before J then nothing of J can be before I. `IIR::Before`$(I, J) > 0 \Rightarrow$ `IIR::Before`$(J, I) = 0$. This does no longer hold for fuzzy sets.



Counterexample for Asymmetry

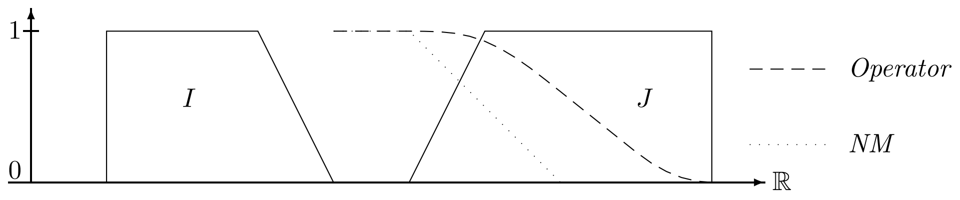
$B(I)$ has a non-zero intersection with J and $B(J)$ has a non-zero intersection with I. Therefore `IIR::Before`$(I, J) > 0$ and `IIR::Before`$(J, I) > 0$.

The fuzzy version of transitivity relates *before*(I, J) and *before*(J, K) with *before*(I, K) in some way. There is such a relation for version 3 of `IIR::Before` and crisp sets: if `IIR::Before`$(I, J) = a$ ($a \cdot 100\%$ of I is before J) and `IIR::Before`$(J, K) = b$ ($b \cdot 100\%$ of J is before K) then one can show that `IIR::Before`$(I, K) = \min(1, a + (b \cdot |I|/|J|))$. Nothing of this kind holds for fuzzy sets.

Version 4 (Nagypál and Motik)

The first picture shows the result of the *before* relation when the interval I is moved along the time axis. A point (t, y) at the dashed and dotted curves is the result of the *before* relation when the positive end of the interval I is moved to position t.

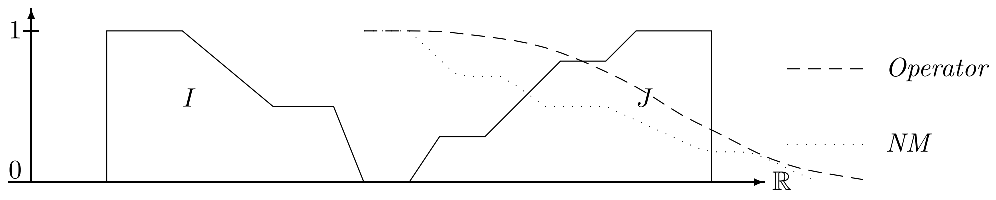
Example 5.3.6 (`IIR::Before`: operator version versus Nagypál and Motik's version)



`IIR::Before`(I, J) *and* `IIR::NMBefore`(I, J)

It may happen that the result is a non-zero fuzzy value even beyond the positive end of J. This is because even if the positive end of I is behind J, there is still some part of I before J. The next example shows that both versions can show the same phenomenon.

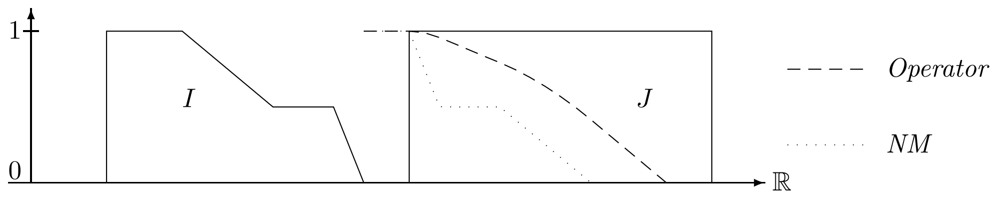
Example 5.3.7 (`IIR::Before`: operator version versus Nagypál and Motik's version)



`IIR::Before`(I, J) *and* `IIR::NMBefore`(I, J)

The last example is with a fuzzy and a crisp interval. It illustrates that in the Nagypál and Motik version of *before* the structure of the right end of I is mapped directly to the structure of the result.

Example 5.3.8 (`IIR::Before`: operator version versus Nagypál and Motik's version)



`IIR::Before`(I, J) *and* `IIR::NMBefore`(I, J)

The examples demonstrate that the Nagypál and Motik version of the *before* relation reveals more about the fine structure of the back end of the first interval I and the front end of the second interval J. The precise relation, however, is not very clear. The operator version, on the other hand, has a more global meaning: the resulting fuzzy value stands for fraction of the first interval which is before the second interval.

5.3.4 Interval–Interval 'meets' Relations

The classical *meets*–relation yields 'true' if the end of the first interval I touches the beginning of the second interval J. This is essentially version 1 of `IIR::Meets` below. Version 1 has again the region parameter `R` (`support`, `core`, `kernel` or `maximum`) which specifies the crisp region of the intervals to be compared. The other three versions in Def. 5.3.9 below are the operator definitions and Nagypál and Motik's version.

Definition 5.3.9 (Interval–Interval 'meets' Relations)

```
1. IIR::Meets(Interval I, Interval J, Region R) =
        case isEmpty(I): false,
             isEmpty(J): false
        else (point(I,right,R) == point(J,left,R))


2. IIR::Meets(Interval I, Interval J, Interval->Interval F,
              Interval->Interval S, Bool simple) =
        case isEmpty(I) or isEmpty(J): 0.0,
             isInfinite(I,right) or isInfinite(J,left): 0.0
        else integrateSymmetric(F(I),S(J),simple)

3. IIR::Meets(Interval I, Interval J, Bool simple) =
        IIR::Meets(I,J,PIR::Finishes[Interval],PIR::Starts[Interval],simple)

4. IIR::NMMeets(Interval I, Interval J) =
        case isEmpty(I) or isEmpty(J): 0.0,
             isInfinite(I,right) or isInfinite(J,left): 0.0
        else Let IN = extend(I,negative) in
             Let JP = extend(J,positive) in
               min(inf(union(IN,JP)),
                   inf(union(complement(IN),complement(JP))))
```

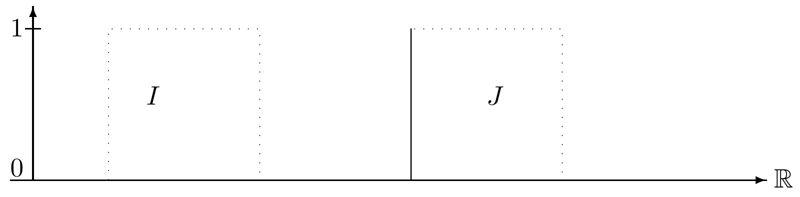
■

Version 2 and 3

The back end of I and the front end of J are relevant for evaluating *meets*(I, J). We can get the back end of I in our fuzzy setting with the point–interval `PIR::Finishes` operator (Def. 5.2.9), and the front end of J with the point–interval `PIR::Starts` operator (Def. 5.2.6, Example 5.2.7). The fuzzy *meets*-relation measures how many points in the back end $F(I)$ of I are in the front end $S(J)$ of J and normalises the value with the maximum possible overlap between $F(I)$ and $S(J)$. Notice that this works only if $|F(I)|$ and $|S(J)|$ are finite. The mathematical definition is therefore $meets_{F,S}(I, J) \stackrel{\text{def}}{=} \int F(I)(t) \cdot S(J)(t)\ dt/N(F(I), S(J))$.

The normalisation factor $N(F(I), S(J)) \overset{\text{def}}{=} \max_a \int F(I)(t-a) \cdot S(J)(t)\ dt$ amounts to a search problem where $F(I)$ is moved along the time axis to find the position for I where the integral becomes maximal. This guarantees that there is a position for I where $meets(I, J) = 1$. Although the implementation of the search procedure is quite efficient, it may be too expensive or it may be unimportant to normalise the *meets*–relation to 1. Therefore version 2 of `IIR::meets` has a parameter `Bool simple` which is passed to the built-in function `integrateSymmetric`. `integrateSymmetric`$(I, J, \texttt{simple})$ computes $\int I(t) \cdot J(t)\ dt / N(I, J)$ where $N(I, J) = min(|I|, |J|)$ if $\texttt{simple} = \texttt{true}$, and $N(I, J) \overset{\text{def}}{=} \max_a \int I(t-a) \cdot J(t)\ dt$ if $\texttt{simple} = \texttt{false}$. The effect of $\texttt{simple} = \texttt{true}$ is that there may be no position of I relative to J where $\texttt{IIR::meets}(I, J, true) = 1$.

Example 5.3.10 (`IIR::meets` for crisp intervals)
The first picture shows the `IIR::meets`*–relation where for crisp intervals the operators F and S have a singular peak. Consequently the* `IIR::meets`*–relation has also a singular peak when the interval I meets J in the crisp sense.*

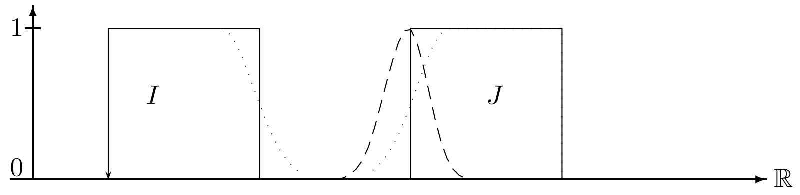


Crisp `IIR::Meets`(I, J) *for Crisp Intervals*

■

The next picture shows the result of the *meets*-relation when the *finishes*- and *starts* operators fuzzify the crisp sets. The dotted lines show the fuzzified crisp sets (with Gaussian fuzzification). The dashed line is again the result of *meets* when the endpoint of I is moved along the time axis.

Example 5.3.11 (Fuzzy `IIR::Meets` for crisp intervals)

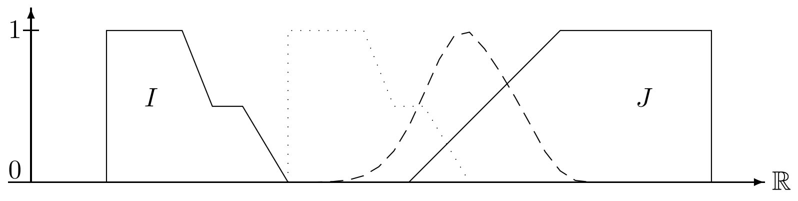


Fuzzy `IIR::Meets` *for Crisp Intervals*

■

Finally we illustrate the fuzzy *meets*-relation with two fuzzy time intervals and the simple point–interval *finishes* and *starts* operators of Example 5.2.7.

Example 5.3.12 (Fuzzy `IIR::Meets` for fuzzy intervals) *The dashed line shows the results of the*
`IIR::Meets`*–relation when the interval I is moved along the time axis. The dotted figure is the position of I where* `IIR::Meets` *is maximal.*



`IIR::Meets`(I, J) *for Fuzzy Intervals*

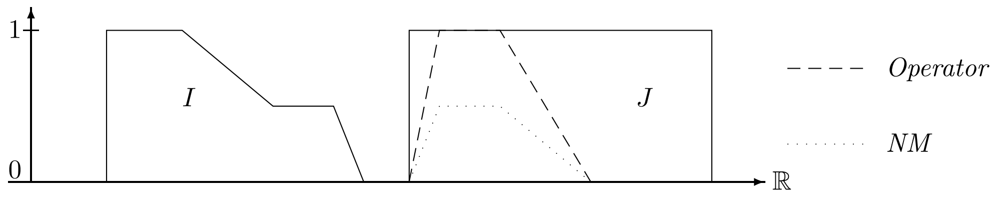
■

Properties of `IIR::Meets`
If the operators F and S have a singular peak then *meets* behaves for crisp relations like the classical crisp *meets*–relation. Therefore the properties of the crisp *meets*–relation hold as well: irreflexivity and asymmetry holds, and transitivity does not hold. If F and S widen the peaks then nothing of this can be predicted any more.

Version 4 (Nagypál and Motik)

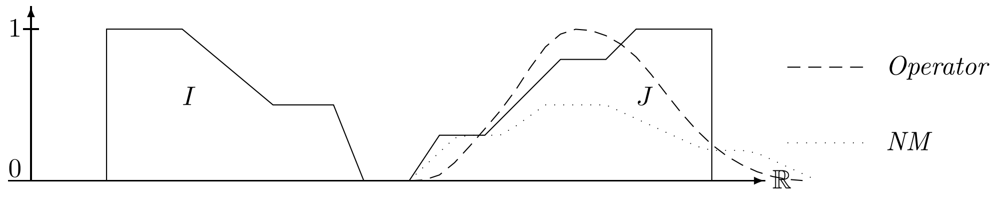
The next examples illustrate that the differences between the operator version of the *meets*–relation and Nagypál and Motik's version are minor. The operator version yields a smoother curve, and the fuzzy values are normalised to 1 as peak value. This would be possible for the Nagypál and Motik version either, but it causes a similar search problem as for the operator version.

Example 5.3.13 (`IIR::Meets`: operator version and Nagypál and Motik's version)



`IIR::Meets`(I, J) *and* `IIR::NMMeets`(I, J)

The dashed and dotted curves indicate again the resulting value of the meets *relation at coordinate t if the interval I is moved such that its positive end is at t.*



`IIR::Meets`(I, J) *and* `IIR::NMMeets`(I, J)

■

5.3.5 Interval–Interval 'overlaps' Relations

The classical relation I *overlaps* J holds if two conditions are fulfilled:
1. a non-empty part I' of I must lie before J, and
2. the rest $(I \setminus I') \neq \emptyset$ must lie inside J.

The generalisation to non-convex intervals concerns the first condition. There are the two possibilities: 1. I' consists of one or more components which lie before J, i.e. there is a gap between the end of I' and the start of J;
2. the last component of I' overlaps with J. This condition is realized in the current implementation of Version 1 of `IIR::Overlaps` below. The 'overlaps' test for non-convex intervals is quite involved. Therefore it is realized as a built-in function `doesOverlap` in GeTS.

Definition 5.3.14 (Interval–Interval 'overlaps' Relations)

```
1. IIR::Overlaps(Interval I, Interval J, Region R) = doesOverlap(I,J,R)

2. IIR::Overlaps(Interval I, Interval J, Interval->Interval E,
                 (Interval*Interval)->Float D) =
         case isEmpty(I) or isEmpty(J) or isInfinite(J,left) : 0.0,
              isInfinite(I,right) and isInfinite(J,right):
                 float(doesOverlap(I,J,support))
              isInfinite(I,right): 0.0,
              isInfinite(J,right): float(doesOverlap(I,J,support))
         else Let EJ = E(J) in
                 (1.0 - D(I,EJ)) * D(I,J) /  NormalizeOverlaps(I,J,EJ,D)

3. IIR::Overlaps(Interval I, Interval J)
```

```
    = IIR::Overlaps(I,J,lambda(Interval K) extend(K,positive),
                    IIR::During[Interval*Interval])

4. IIR::NMOverlaps(Interval I, Interval J) =
        if(isEmpty(I) or isEmpty(J)) then 0.0
        else Let IL = extend(I,negative) in
             Let IR = extend(I,positive) in
             Let JL = extend(J,negative) in
             Let JR = extend(J,positive) in
                min(sup(intersection(IR,complement(JR))),
                    sup(intersection(IL,JR)),
                    sup(intersection(complement(IL),JL)))
```

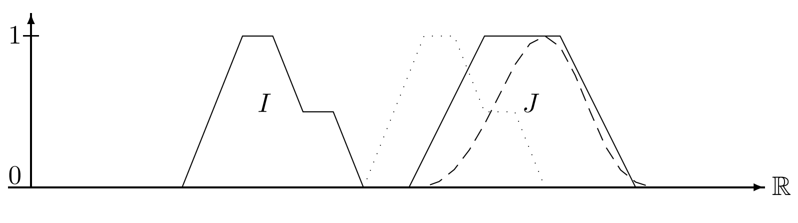
■

Versions 2 and 3

Version 2 needs two operators as parameters for `IIR::Overlaps`. The parameter `E` should be a function that extracts the front part of the interval J and extends it to infinity. `extend(J,positive)` is the natural choice. The operator `D` should be one of the *during*–operators.

Versions 2 of `IIR::Overlaps` for Finite Intervals
The first condition, a non-empty part I_1 of I must lie before J, is encoded in the factor $1 - D(I, E(J))$ $D(I, E(J))$ measures the part of I which is after the front part of J. $1 - D(I, E(J))$ then measures the part of I which is before the front part of J. This factor is multiplied with $D(I, J)$ which corresponds to the second condition. It measures to which degree I is contained in J. The product is normalised with $\max_a((1 - D(shift(I, a), E(J))) \cdot D(shift(I, a), J))$ The normalisation factor corresponds to the maximal possible overlap when I is shifted along the time axis. This guarantees that there is a position for I where `IIR::Overlaps`$(I, J) = 1$. The normalisation factor is computed by the built-in function `NormalizeOverlaps`.

Example 5.3.15 (`IIR::Overlaps` for fuzzy intervals) *This example shows the result of the* `IIR::Overlaps` *relation where the standard* during *operator is used (with the identity function as point–interval* during *operator).*



`IIR::Overlaps`(I, J)

The dashed line represents the result of the overlaps relation for a time coordinate t where the positive end of the interval I is moved to t. The dotted figure indicates the interval I moved to the position where `IIR::Overlaps`(I, J) *becomes maximal.* ■

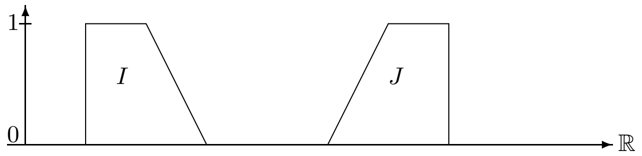
The normalisation factor $\max_a((1 - D(I'(a), E(J))) \cdot D(I'(a), J))$ causes again a search problem. As one can see in the above example the search space is usually very simple (if the intervals are not too exotic). There is only one global maximum and no local maxima. Therefore standard hill climbing is an efficient search method in this case.

Versions 2 of `IIR::Overlaps` for Infinite Intervals
If the interval J is negative infinite then there cannot be anything before J. Therefore `IIR::Overlaps`$(I,J) = 0$. In the other infinite cases we compute the crisp overlaps relation for the support of I and J.

Properties of the *overlaps* relation:
The crisp *overlaps* relation is irreflexive, asymmetric and not transitive. For version 3 of `IIR::Overlaps` and finite fuzzy sets I which are crisp at the left side we have $D(I, E(I)) = 1$. Therefore,

$$\texttt{IIR::Overlaps}(I,I) = (1 - D(I, E(I)) \cdot D(I,I)/N'(I,J) = 0$$

holds. If I is fuzzy at the left side then $D(I, E(I)) < 1$ and $D(I,I) > 0$, and therefore `IIR::Overlaps`$(I,I) > 0$.



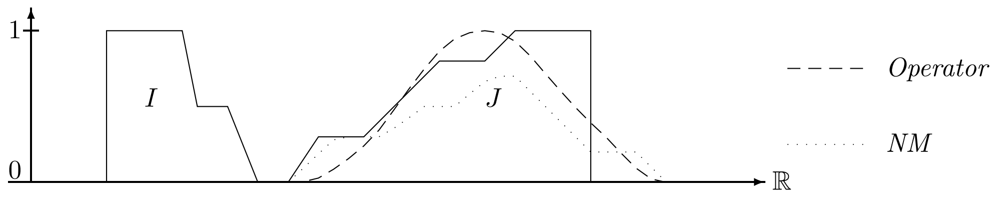
`IIR::Overlaps`$(I,I) = 0$ *and*
`IIR::Overlaps`$(J,J) > 0$

For crisp intervals I and J we have a property which corresponds to asymmetry: `IIR::Overlaps`$(I,J) > 0 \Rightarrow$ `IIR::Overlaps`$(J,I) = 0$. This is because `IIR::Overlaps`$(I,J) > 0$ means that a part of I is before J. Therefore no part of J can be before I. If I and J are fuzzy, we can have `IIR::Overlaps`$(I,J) > 0$ and `IIR::Overlaps`$(J,I) > 0$.

Version 4 (Nagypál and Motik)

The first example below for the *overlaps* relation shows a structural similarity between the operator version and Nagypál and Motik's version. Again, Nagypál and Motik's version is not normalised.

Example 5.3.16 (`IIR::Overlaps`: operator version versus Nagypál and Motik's version)

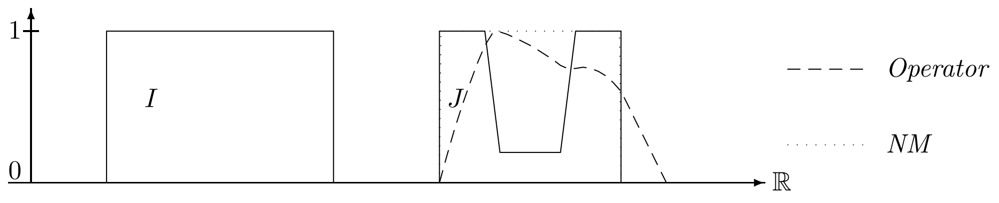


`IIR::Overlaps`(I,J) *and* `IIR::OverlapsNM`(I,J)

The dashed and dotted lines show the result of `IIR::Overlaps` *at a point t when the interval I is shifted such that its endpoint is at t.* ■

The next example demonstrates that Nagypál and Motik's version of the *overlaps* relation is not sensitive to the internal structure of the intervals. The interval J is treated like its crisp hull.

Example 5.3.17 (`IIR::Overlaps`: **operator version versus Nagypál and Motik's version)**



`IIR::Overlaps`(I, J) *and* `IIR::OverlapsNM`(I, J)

■

5.3.6 Interval–Interval 'starts' Relations

The crisp I *starts* J-relation has two conditions:
1. the start point of I and the start point of J are identical, and
2. I is a subset of J.

These conditions can be generalised straightforwardly to non-convex intervals. They are checked in version 1 of the `IIR::Starts`–relation below.

Definition 5.3.18 (Interval–Interval 'starts' Relations)

```
1. IIR::Starts(Interval I, Interval J, Region R) =
        if(isEmpty(I) or isEmpty(J)) then false
        else (point(I,left,R) == point(J,left,R)) and isSubset(I,J,R)

2. IIR::Starts(Interval I, Interval J, Interval->Interval S,
              (Interval*Interval)->Float D, Bool simple) =
        case isEmpty(I) or isEmpty(J) or isInfinite(J,left) or
             (isInfinite(I,left) xor isInfinite(J,left)): 0.0,
             isInfinite(I,left) and isInfinite(J,left) : D(I,J)
        else integrateSymmetric(S(I),S(J),simple)*D(I,J)

3. IIR::Starts(Interval I, Interval J, Bool simple) =
      IIR::Starts(I,J,PIR::Starts[Interval], IIR::During[Interval*Interval],simple)

4. IIR::NMStarts(Interval I, Interval J) =
        if(isEmpty(I) or isEmpty(J)) then 0.0
        else Let IR = extend(I,positive) in
             Let JR = extend(J,positive) in
                min(inf(union(complement(IR),JR)),
                    inf(union(IR,complement(JR))),
                    sup(intersection(complement(extend(I,negative)),
                                         extend(J,negative))))
```

■

Versions 2 and 3

The two conditions for the crisp *starts*–relation can be reformulated for the operator version. The conditions are turned into a product of the overlap between the two starting sections of I

and J, and the *during*(I, J)-relation:

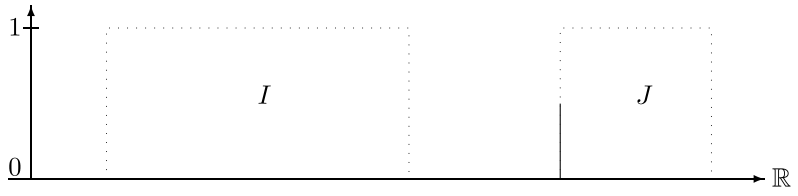
$$\textit{starts}(I,J) \overset{\text{def}}{=} \frac{\int S(I)(t)\cdot S(J)(t)\ dt}{N(S(I),S(J))}\cdot D(I,J)$$

The first factor checks the first condition: the starting part $S(I)$ of I should coincide with the starting part $S(J)$ of J. This value is normalised to the maximal possible overlap of the starting parts. The assumption here is that if I move I along the time axis, there should be a position of I where I definitely starts J, and where therefore the fuzzy value should be 1. The second factor checks whether I is a subset of J. This factor need not be 1 if I is larger than J. Therefore the result of *starts*(I, J) can be < 1 regardless of the position of I.

The extreme cases are:

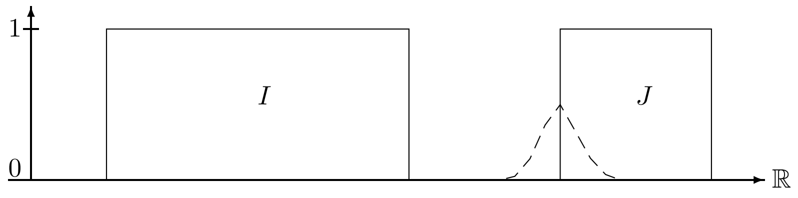
- if I or J are empty then *starts*(I, J) must be 0;
- if one of I and J is negative infinite then they can't have the same starting point. Therefore *starts*(I, J) must be 0 again;
- if both are negative infinite then we can assume that they have the same starting point, and therefore only the second condition *during*(I, J) must be checked;
- it does not matter whether I or J are positive infinite because only the finite starting sections of I and J count.

Example 5.3.19 (`IIR::Starts` for crisp intervals)
The first picture shows the case where the identity operator is used for D. If the front end of I is moved along the time axis we get a single peak when it meets J. The peak, however, is only 0.5 high because I is twice as large as J.



`IIR::Starts`

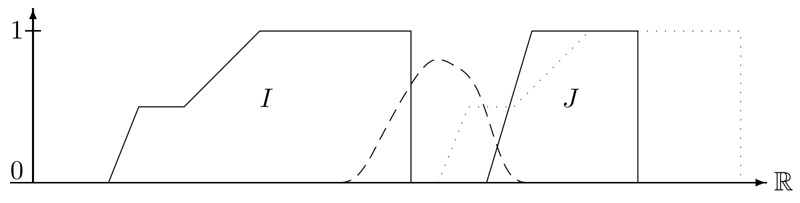
The next picture shows a fuzzified `IIR::Starts`*–relation. The dashed line shows the value of* starts(I, J) *for a position t where the* front end *of I is moved to t. The crisp intervals are fuzzified in the same way as in Example 5.3.10. The peak is broader, but the maximum is still at 0.5 because I is twice as large as J.*



Fuzzified `IIR::Starts`*–relation*

■

Example 5.3.20 (`IIR::Starts`–Relation for fuzzy intervals) *The next figure shows the application of the same* `IIR::Starts`*–relation as in the first picture of the above example to fuzzy intervals. The dashed line is again the result of the* `IIR::Starts`*–relation. The dotted figure shows the position of the interval* I *where* `IIR::Starts`(I, J) *is maximal.*



`IIR::Starts`*(I,J)*

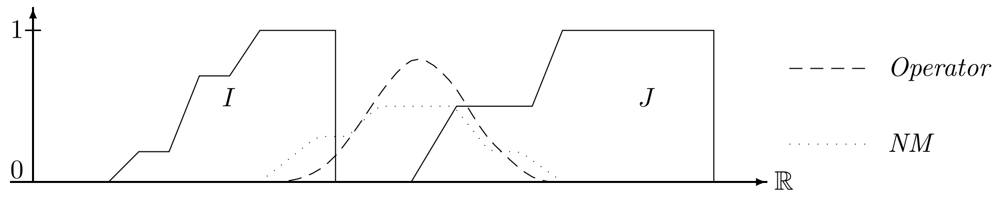
■

Properties of the `IIR::Starts`(I,J)–relation:
The classical *starts* relation for crisp intervals is reflexive, antisymmetric and transitive. If we consider only the cases `IIR::Starts`$(I, J) = 0$ or `IIR::Starts`$(I, J) = 1$ where I and J are crisp intervals then it behaves like the classical starts relation. The same properties hold, in particular reflexivity. For fuzzy intervals, however, we have $0 <$ `IIR::Starts`$(I, I) < 1$. No kind of fuzzy antisymmetry holds for `IIR::Starts` on fuzzy intervals. A fuzzy version of transitivity does also not hold. The reason is that, although the starting sections of I and J may overlap, and the starting sections of J and K may overlap, this does not imply that the starting sections of I and K overlap.

Version 4 (Nagypál and Motik)

This version shows close structural similarities if the left sides of the intervals are not too exotic.

Example 5.3.21 (`IIR::Starts`(I, J) and `IIR::StartsNM`(I, J))

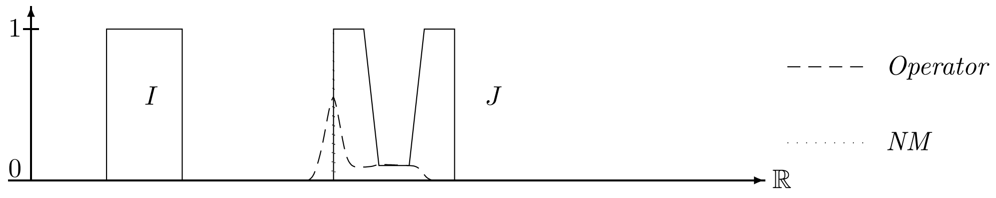


`IIR::Starts`(I, J) *and* `IIR::StartsNM`(I, J)

The dashed and dotted lines in this picture show the resulting values of starts *relation at a point t if the* left *end of the interval I is moved to t.* ■

The Nagypál and Motik *starts*–relation in the next picture has a singular peak of maximal high for the case that the left end of the interval I coincides with the left end of the interval J, although I is not contained in J. The operator version with standard parameters has also a singular peak there, but its hight is only 0.55, which indicates that only 55% of I are contained in J. The dashed line in the picture shows the results of the operator version of *starts*, where the starting sections of I and J are widened by a fuzzification operator.

Example 5.3.22



`IIR::Starts`(I, J) *and* `IIR::StartsNM`(I, J)

■

5.3.7 Interval–Interval 'finishes' Relations

The *finishes*–relation is the mirror image of the *starts*–relation. Therefore we list only the definitions.

Definition 5.3.23 (Interval–Interval 'finishes' Relations)

```
1. IIR::Finishes(Interval I, Interval J, Region R) =
        case isEmpty(I): false,
             isEmpty(J): false
        else (point(I,right,R) == point(J,right,R)) and isSubset(I,J,R)

2. IIR::Finishes(Interval I, Interval J, Interval->Interval S,
              (Interval*Interval)->Float D, Bool simple) =
        case isEmpty(I) or isEmpty(J) or isInfinite(J,left) or
                   (isInfinite(I,right) xor isInfinite(J,right)): 0.0,
             isInfinite(I,right) and isInfinite(J,right) : D(I,J)
        else integrateSymmetric(S(I),S(J),simple)*D(I,J)

3. IIR::Finishes(Interval I, Interval J, Bool simple) =
      IIR::Finishes(I,J,PIR::Finishes[Interval], IIR::During[Interval*Interval],simple)

4. IIR::NMFinishes(Interval I, Interval J) =
        if (isEmpty(I) or isEmpty(J)) then 0.0
        else Let IL = extend(I,negative) in
             Let JL = extend(J,negative) in
                min(inf(union(IL,complement(JL))),
                    inf(union(complement(IL),JL)),
                    sup(intersection(extend(I,positive),
                                     complement(extend(J,positive)))))
```
■

5.3.8 Interval–Interval 'during' Relations

The crisp version of the *during*–relation needs only to check whether an interval I is a subset of an interval J. If I and J are not convex, this is a non-trivial task, but can still be done in linear time with a sweep line algorithm. Therefore it is realized by the built-in function `isSubset` in version 1 of `IIR::During` below.

Definition 5.3.24 (Interval–Interval 'during' Relations)

```
1. IIR::During(Interval I, Interval J, Region R) = isSubset(I,J,R)

2. IIR::During(Interval I, Interval J, Interval->Interval D) =
        case isEmpty(I): 1.0,
             isEmpty(J): 0.0,
             isInfinite(I):
                Let iNeg=member(point(I,left,support),I) in
                Let iPos=member(point(I,right,support),I) in
                    (iNeg * member(point(J,left,support),I) +
                     iPos * member(point(J,right,support),I)) /
                        (iNeg + iPos)
        else integrateAsymmetric(I,D(J))

3. IIR::During(Interval I, Interval J) =
        IIR::During(Interval I, Interval J, lambda(Interval K) K);

4. IIR::NMDuring(Interval I, Interval J) =
        case isEmpty(I): 1.0,
             isEmpty(J): 0.0
        else min(sup(intersection(complement(extend(I,positive)),extend(J,positive))),
                 sup(intersection(complement(extend(I,negative)),extend(J,negative))))
```
■

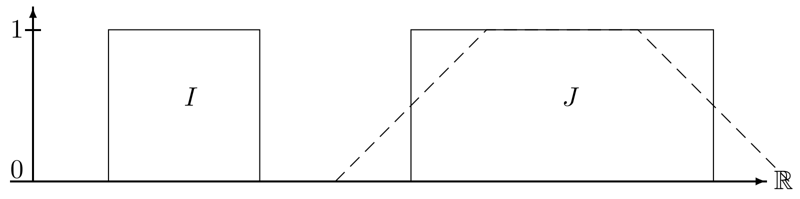
Version 2 and 3

`IIR::During` for Finite Intervals
The operator version of *during* averages the point–interval *during* relation D by integrating over the interval I. The basic formula is therefore

$$during_D(I,J) \overset{\text{def}}{=} \frac{\int I(t)\cdot D(J)(t)\ dt}{|I|}$$

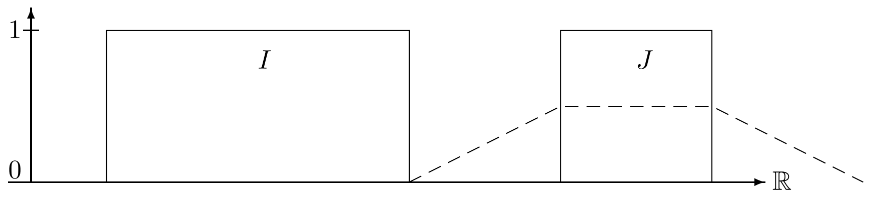
This is realized with a parameter `D` in version 2 of `IIR::During`, and with the identity function for `D` in version 3. `IIR::During`(I,J) measures to what degree I is contained in J. The normalisation factor is $|I|$ because if I is larger than J then `IIR::During`(I,J) should definitely be smaller than 1.

Example 5.3.25 (`IIR::During` for crisp intervals)
The dashed line in the picture below shows the result of the `IIR::During`*-relation for a coordinate t when the* middle point *of the interval I is moved to t.*



`IIR::During`(I,J) *for Crisp Intervals*

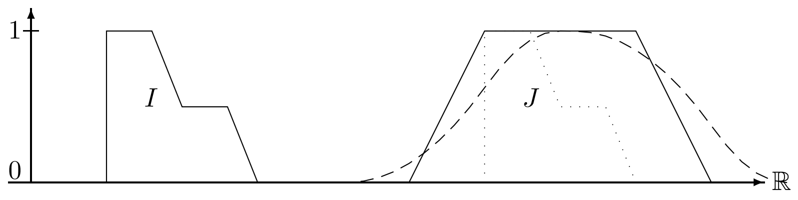
The interval I in the next picture is larger than J such that `IIR::During`(I,J) *never rises to 1.*



`IIR::During`(I, J) *for Crisp Intervals*

■

Example 5.3.26 (`IIR::During` for fuzzy intervals)
The dashed line shows again the result of the `IIR::During`*-relation when the middle point of I is moved along the time axis. The dotted figure indicates the position of I where* `IIR::During`(I, J) *is maximal.*



`IIR::During`(I, J) *for Fuzzy Intervals*

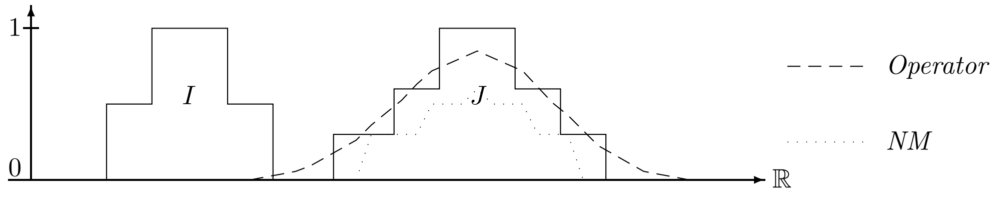
■

`IIR::During` for Infinite Intervals
The formula $\int I(t) \cdot D(J)(t)\ dt/|I|$ cannot be evaluated if I is an infinite interval. Instead one can compute $\lim_{a\to\infty} \int_{-a}^{+a} I(t) \cdot D(J)(t)\ dt / \int_{-a}^{+a} I(t)\ dt$. If the limes is calculated analytically we obtain $(I(-\infty) \cdot J(-\infty) + I(+\infty) + J(+\infty))/(I(-\infty) \cdot I(+\infty))$. This formula is used for version 2 and 3 of `IIR::During` on infinite intervals I.

Version 4 (Nagypál and Motik)

The next two examples show a comparison of the *during*–relations. Compared to the operator version, the Nagypál and Motik version has a much narrower graph and it is less smooth.

Example 5.3.27 (`IIR::During`: operator version versus Nagypál and Motik's version)

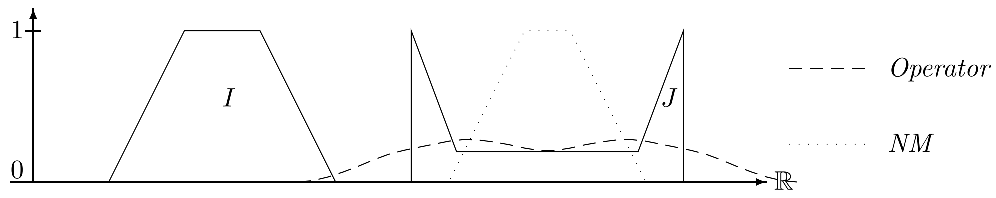


`IIR::During`(I, J) *and* `IIR::NMDuring`(I, J)

The dashed and dotted lines indicate the values of the during relation at coordinate t if the middle point of the interval I is moved to t. ■

The interval J in the next example is treated like its crisp hull by the Nagypál and Motik version. Therefore there is quite a difference to the operator version.

Example 5.3.28 (`IIR::During`: operator version versus Nagypál and Motik's version)



`IIR::During`(I, J) *and* `IIR::NMDuring`(I, J)

The dashed and dotted lines indicate again the values of the during relation at coordinate t if the middle point of the interval I is moved to t. ■

5.3.9 Interval–Interval 'equals' Relations

An interval I equals an interval J if I is a subset of J and vice versa. This is what is tested in version 1 below.

Definition 5.3.29 (Interval–Interval 'equals' Relations)

```
1. IIR::Equals(Interval I, Interval J, Region R) =
       isSubset(I,J,R) and isSubset(J,I,R)

2. IIR::Equals(Interval I, Interval J, (Interval*Interval)->Float D) =
        D(I,J) * D(J,I)

3. IIR::Equals(Interval I, Interval J) = IIR::During(I,J) * IIR::During(J,I)

4. IIR::NMEquals(Interval I, Interval J) =
        case isEmpty(I) : float(isEmpty(J)),
             isEmpty(J) : 0.0
        else Let IL = extend(I,negative) in
             Let IR = extend(I,positive) in
             Let JL = extend(J,negative) in
             Let JR = extend(J,positive) in
                min(inf(union(IL,complement(JL))),
                    inf(union(complement(IL),JL)),
                    inf(union(complement(IR),JR)),
                    inf(union(IR,complement(JR))))
```
■

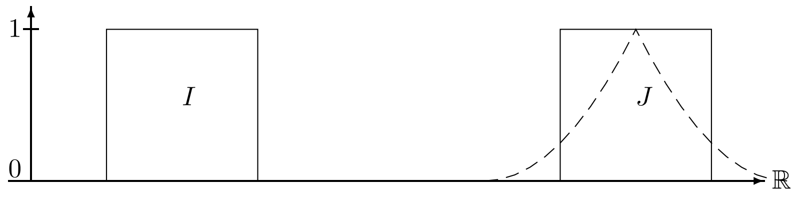
Version 2 and 3

Version 2 takes an arbitrary binary interval operator D, usually a *during*–operator and computes

$$equals_D(I, J) \overset{\text{def}}{=} D(I, J) \cdot D(J, I).$$

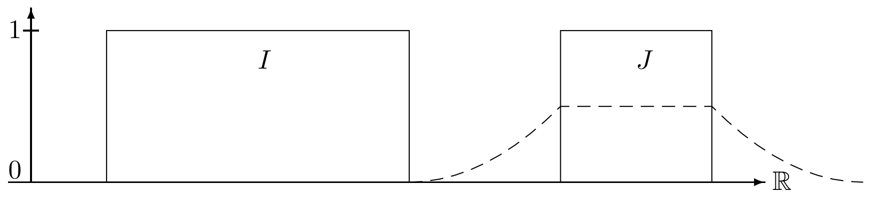
Version 3 uses `IIR::During` for the parameter D.

Example 5.3.30 (`IIR::Equals` for crisp intervals)
The first picture shows the `IIR::Equals`*–relation for similar intervals. If I is moved on top of J then* `IIR::Equals`$(I, J) = 1$



`IIR::Equals` *for Similar Intervals*

I and J in the next figure are not equal. Therefore equals(I, J) *never rises to 1.*



`IIR::Equals` *for Different Intervals*

■

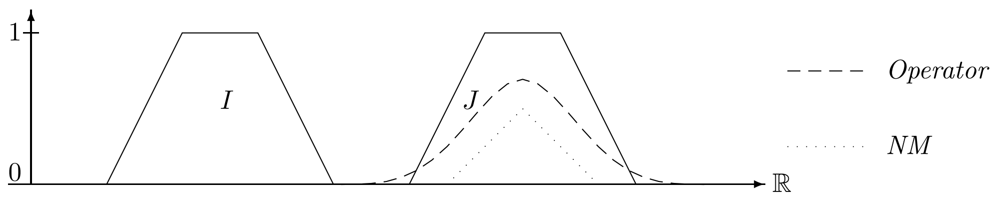
Properties of the `IIR::Equals` relation:
The classical *equals*–relation is an equivalence relation. `IIR::Equals`$(I, J) = 1$ implies that I and J are crisp and moreover $I = J$ (see the corresponding remark about `IIR::During`$(I, J) = 1$). Therefore if we consider only the case `IIR::Equals`$(I, J) = 1$ then `IIR::Equals` is also an equivalence relation.

Since `IIR::During`$(I, I) > 0$ we also have `IIR::Equals`$(I, I) > 0$ for non-empty fuzzy sets. By the very definition of `IIR::Equals` we have `IIR::Equals`$(I, J) =$ `IIR::Equals`(J, I), the strongest form of fuzzy symmetry. Any form of fuzzy transitivity does not hold. It is easy to construct examples where `IIR::Equals`$(I, J) > 0$ and `IIR::Equals`$(J, K) > 0$ and `IIR::Equals`$(I, K) = 0$. This is because although `IIR::Equals`$(I, J) > 0$ requires an overlap between I and J, and `IIR::Equals`$(J, K) > 0$ requires an overlap between J and K, there need not be an overlap between I and K.

Version 4 (Nagypál and Motik)

The behaviour of the *equals* relations are quite similar to the behaviour of the *during* relations because they essentially consist of two *during* relations. The two curves in the first example below do not rise to 1, although the shapes of the two intervals are identical. This is, because the relations do not compare the shapes of the intervals, but they rely on the meaning of the *during* relation.

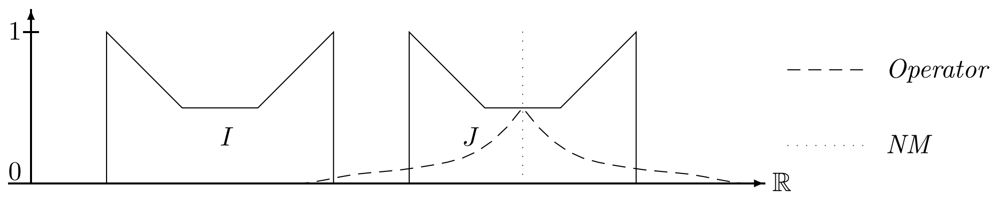
Example 5.3.31 (`IIR::Equals`: operator version and Nagypál and Motik's version)



`IIR::Equals`(I, J) *and* `IIR::Equals`(I, J) ■

The Nagypál and Motik version of the *equals* relation in the next example has a singular peak when the two intervals match exactly. The reason is again that the Nagypál and Motik version treat the intervals in this case like their crisp hulls.

Example 5.3.32 (`IIR::equals`: operator version and Nagypál and Motik's version)



`IIR::Equals`(I, J) *and* `IIR::Equals`(I, J) ■

5.3.10 Summary: Operator Version versus Nagypál and Motik's Relations

The resulting values of the operator version (with standard parameters) and the Nagypál and Motik version of the different relations show a similar structure if the intervals are 'crisp like' intervals, i.e. if they consist of a single component which is more or less monotonic, and has no internal structure. The Nagypál and Motik version follows the rising and falling parts of the intervals more directly, whereas the operator version smoothes the curves. Since the operator versions integrate over the corresponding point-interval relations, they have a more intuitive meaning than the Nagypál and Motik version.

The differences between the two versions are more obvious for crisp intervals, where the Nagypál and Motik versions deliberately behave like the pure crisp relations. The operator versions can return non-trivial fuzzy values is these cases. The differences become also more obvious when the intervals have an internal structure, or when they consist of several components. The internal structure is completely ignored by the Nagypál and Motik versions. This is not the case for the operator version.

Transitivity Table? Allen's interval relations for crisp intervals are related in particular ways. For example, if *I starts J* holds then *I during J* must hold as well. All the relationships between the different interval relations can be collected in a *transitivity table*. The transitivity table can then be used for constraint propagation algorithms. A systematic investigation of the relationships between fuzzy interval–interval relations has not been done yet. The guess is that not many relationships hold, and therefore the transitivity table may not help much for a constraint propagation algorithm.

5.3.11 Until

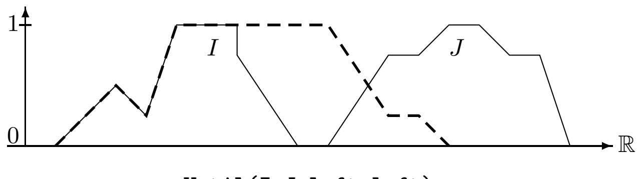
The 'Until' operator is known from temporal logics [17]. φ *Until* ψ in a temporal logic usually means 'eventually ψ holds and φ holds *until* this time point. Since 'Until' is quite useful, we show two GeTS versions of 'Until' where φ and ψ are not formulae, but fuzzy time intervals. With this Until operator we can model expressions like 'from early morning until late night', where 'early morning' and 'late night' are concrete fuzzy intervals. An expression like this is ambiguous. It can be interpreted as 'from the beginning of early morning until the end of late night' or 'from the beginning of early morning until the beginning of late night', and there are two more possibilities. All four combinations are provided. The first version in Def. 5.3.33 below uses the concrete operators `extend`, `complement` and `intersection`. The second version accepts these operators as extra parameters.

Definition 5.3.33 (Until)

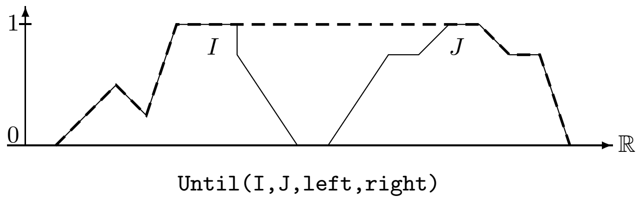
```
1. Until(Interval I, Interval J, Side s1, Side s2) =
   if (s1 == left) then
      (if (s2 == left) then
            intersection(extend(I,positive),complement(extend(J,positive)))
       else intersection(extend(I,positive),extend(J,negative)))
   else
      (if (s2 == left) then
            intersection(complement(extend(I,negative)),complement(extend(J,positive)))
       else intersection(complement(extend(I,negative)),extend(J,negative)));
```

```
2. Until(Interval I, Interval J, Side s1, Side s2, (Interval*Interval)->Interval Ints,
     Interval->Interval Ep,  Interval->Interval En, Interval->Interval C) =
        if (s1 == left) then
           (if (s2 == left) then Ints(Ep(I),C(Ep(J)))
                             else Ints(Ep(I),En(J)))
        else
           (if (s2 == left) then Ints(C(En(I)),C(Ep(J)))
                             else Ints(C(En(I)),En(J)));
```

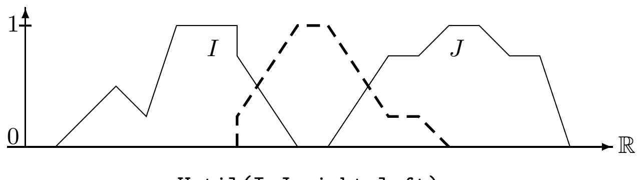
The next four figures show the four cases for the `Until` operator when two single non-overlapping fuzzy intervals are involved.



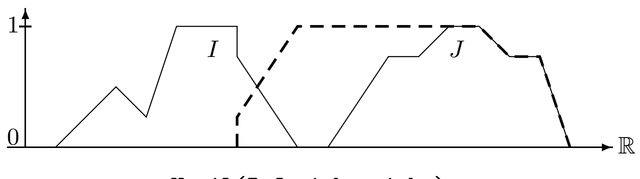
`Until(I,J,left,left)`



`Until(I,J,left,right)`



`Until(I,J,right,left)`

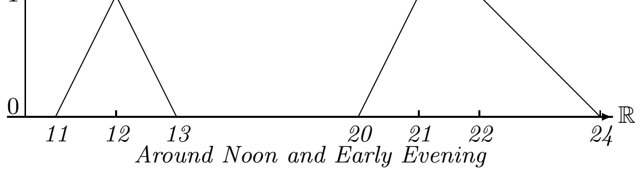


`Until(I,J,right,right)`

The next example shows a concrete application of the second version of `Until` with the following call:

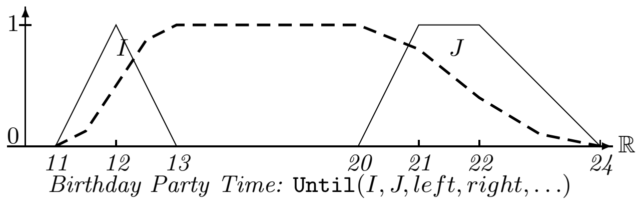
```
Until(I, J, left, right, lambda(Interval K, Interval L) intersection(K,L),
      lambda(Interval K) integrate(K,positive),
      lambda(Interval K) integrate(K,negative),
      lambda(Interval K) complement(K)).
```

Example 5.3.34 (Birthday Party Time) *The Until operator can be used in more sophisticated ways. Consider a database about, say, the institute's birthday parties. It may contain the entry that the birthday party for the director took place 'from around noon until early evening' of 20/7/2003. 'Around noon' is a fuzzy notion and 'early evening' is a fuzzy notion. Suppose, we have a formalisation of 'around noon' and 'early evening' as the following fuzzy sets:*



Around Noon and Early Evening

What is now the duration of the birthday party? It must obviously also be a fuzzy set. The fuzzy value of the birthday party duration at a time point t is 1 if the probability that the party started before t is 1 and the probability that the party ended after t is also 1. Therefore the fuzzy value at point t is computed by integrating over the probabilities of the start points and the end points. The resulting fuzzy set is:



Birthday Party Time: $\texttt{Until}(I, J, left, right, \ldots)$

The dashed curve may, for example, represent the percentage of people at the party at a give time. ■

5.4 Summary

Different versions of point–interval and interval–interval relations for fuzzy intervals have been presented in this paper. They are defined in the GeoTemporal Specification language (GeTS). The point–interval relations are actually defined as functions which map intervals to intervals. For example, the point–interval *before* function maps an interval I to the interval of all points before I. Three different versions of interval–interval relations are defined. The first version works like the crisp relations by taking crisp regions of the intervals. The second version, the 'operator version' uses suitable point–interval relations and extends the point to an interval by averaging (integrating) over the second interval. The third version is taken from the literature [25]. The effect of the different versions is illustrated graphically with concrete examples. The properties of the operator versions are investigated as far as it is possible. Since the operator version is extremely flexible and can be instantiated in many different ways, an exhaustive investigation of its properties is almost impossible. Therefore, only a few important cases are considered. The operator version is compared with Nagypál and Motik's version by applying them to typical examples. A more general theoretical comparison is not really feasible. The definitions of the point–interval and interval–interval relations are available and can be loaded into the CTTN system.

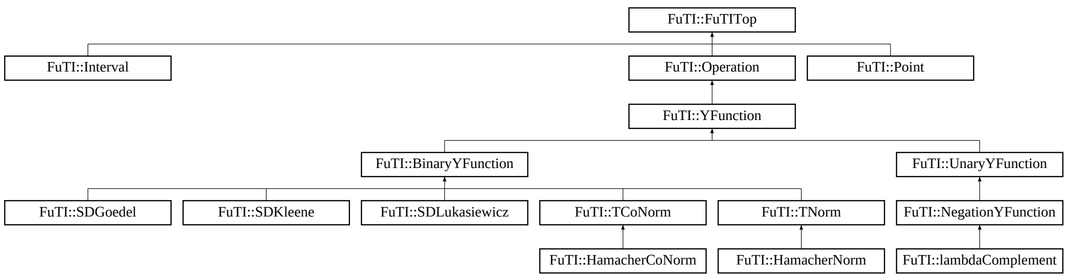
Appendix A

The FuTI–Module

A.1 The FuTI—interface

This is a very brief overview of the FuTI–interface. A much more detailed documentation is available as doxygen generated html and latex files.

The FuTI–interface consists of the main classes *Point*, *Interval*, *Operation* and *YFunction*. Additionally there are a number of auxiliary classes. The top class for all classes in the CTTN system is *FuTITop*. In addition there is a namespace *Service* which is imported from the CTTN system. This class contains all the general purpose functions which cannot be associated to a particular component system. The *YFunction* has also a number of subclasses. The complete class hierarchy is as follows:



For the main classes we list the constructor methods, the main public methods and explain briefly what they do. The syntax we use in this section is simplified C++ or Java. The precise syntax is of course in the corresponding header or class files.

CX is the datatype of the x-coordinates (long long integers or multiple precision integers) and CY is the datatype of the y-coordinates (typically unsigned short integers). CX and CY are compiler options.

Many of the methods represent partial functions. FuTI has a DEBUG mode (compiler option) where the necessary preconditions are checked and an error is thrown when the preconditions are not met. If the DEBUG mode is turned off, only the errors which can be caused by data and not by program errors are still caught.

A.1.1 Points

This class represents 2-dimensional points with coordinates of type CX and CY.

Constructors

`Point(CX x, CY y)`
constructs a point from `x` and `y`-coordinates.

`Point(string s)`
reconstructs a point from a string representation "x,y".

Predicates

`bool p.leftturn(Point q, Point r)`
true if `p` → `q` → `r` is a left turn or collinear. (Def. 2.3.4)

`bool p.leftturnProper(Point q, Point r)`
true if `p` → `q` → `r` is a proper left turn.

`bool p.rightturn(Point q, Point r)`
: true if $\mathtt{p} \rightarrow \mathtt{q} \rightarrow \mathtt{r}$ is a right turn or collinear.

`bool p.rightturnProper(Point q, Point r)`
: true if $\mathtt{p} \rightarrow \mathtt{q} \rightarrow \mathtt{r}$ is a proper right turn.

`bool p.collinear(Point q, Point r)`
: true if $\mathtt{p} \rightarrow \mathtt{q} \rightarrow \mathtt{r}$ is collinear (Def. 2.3.3)

`bool p.collinear(Point q, Point r, Point s)`
: true if the line segment $(\mathtt{p}, \mathtt{q})$ is collinear with the line segment $(\mathtt{r}, \mathtt{s})$.

`bool p.between(Point q, Point r)`
: true if `p` is between `q` and `r`.

`bool p.betweenProper(Point q, Point r)`
: true if `p` is between `q` and `r`, but different to `q` and `r`.

`bool p.intersects(Point q, Point r, Point s)`
: true if the line $(\mathtt{p}, \mathtt{q})$ intersects the line $(\mathtt{r}, \mathtt{s})$.

`bool p.intersectsProper(Point q, Point r, Point d)`
: true if the line $(\mathtt{p}, \mathtt{q})$ intersects the line $(\mathtt{r}, \mathtt{s})$, but does not only touch it. (Def. 2.3.5)

Computations

`CX p.intersection(Point q, Point r, Point s)`
: computes the intersection point for the line segments $(\mathtt{p}, \mathtt{q})$ and $(\mathtt{r}, \mathtt{s})$. An error is thrown if the line segments do not intersect! (Def. 2.3.6)

`float p.lineY(Point q, CX x)`
: computes for the line crossing `p` and `q` the y-value at point `x`. An error is thrown if the line is vertical. (Def. 2.3.7)

`CX p.lineX(Point q, CY y)`
: computes for the line crossing `p` and `q` the x-value at point `y`. An error is thrown if the line is horizontal. (Def. 2.3.8)

`CX p.area2(Point q)`
: computes the area below the line segment $(\mathtt{p}, \mathtt{q})$. (Def. 2.3.9)

`float p.area2(Point q, CX x1, CX x2)`
: computes the area below the line segment $(\mathtt{p}, \mathtt{q})$ from `x1` until `x2`. It throws an error if the line is vertical and $\mathtt{x1} \neq \mathtt{x2}$ (Def. 2.3.9)

`CX p.area2X(Point q, float a)`
: computes the x-coordinate x such that the area below the line segment $(\mathtt{p}, \mathtt{q})$ from `p` until x is just `a`. An error is thrown if there is not enough area below the line segment. (Def. 2.3.10)

`float p1.integrate(Point p2, Point q1, Point q2, CX x1, CX x2)`
: computes $\int_{\mathtt{x1}}^{\mathtt{x2}} l_1(x) \cdot l_2(x)\, dx$ where l_1 is the line crossing `p1` and `p2` and l_2 is the line crossing `q1` and `q2`. It throws an error if one of the lines is vertical. (Def. 2.3.12)

A.1.2 Intervals

The Interval class manages and manipulates fuzzy temporal intervals (Sec. 2.3.2). The intervals are represented by their envelope polygons (Def. 2.3.15).

Constructors

`Interval()`
: constructs an empty interval.

`Interval(Point p)`
: constructs an interval with a single point `p`.

`Interval(CX x, CY y)`
: constructs an interval with a single point (`x`, `y`).

`Interval(CX a, CX b)`
: constructs a crisp interval [`a`, `b`[.

`Interval(vector<Point> points)`
: constructs an interval with a vector of points.

`Interval(string s)`
: constructs an interval from a string representation $[x_1, y_1\ x_2, y_2\ ...[$.

Adding and Removing Points.

`void I.push_back(Point p)`
: adds the point `p` to the end of the polygon. It throws an error if `p`.x is before the last point in the polygon. (Def. 2.3.18)

`void I.push_back(CX x, CY y)`
: adds the point `x`, `y` to the end of the polygon. It throws an error if `x` is before the last point in the polygon. (Def. 2.3.18)

`void I.pop_back()`
: removes the last point from the polygon. It does nothing on empty polygons. (Def. 2.3.18)

Simple Properties of the Intervals

`Point I.front()`
: returns the leftmost point and throws an error if `I` = ().

`Point I.back()`
: returns the rightmost point and throws an error if `I` = ().

`CX I.frontX()`
: returns the leftmost x-coordinate and throws an error if `I` = ().

`CX I.backX()`
: returns the rightmost x-coordinate and throws an error if `I` = ().

`CY I.frontY()`
: returns the leftmost y-coordinate and throws an error if `I` = ().

`CY I.backY()`
: returns the rightmost y-coordinate and throws an error if `I` = ().

`bool I.isNegInfinite()`
: returns true if the interval is negative infinite.

`bool I.isPosInfinite()`
: returns true if the interval is positive infinite.

`bool I.isInfinite()`
: returns true if the interval is infinite.

`bool I.isEmpty()`
: return true if the polygon is empty.

`bool I.isNonempty()`
: returns true if the polygon is not empty.

`int I.nPoints()`
: returns the number of points in the polygon.

`int I.isCrisp()`
: returns true if the polygon is non-empty, finite and crisp.

`int I.isCrisp(bool front)`
: checks for crispness at the left/right side of the polygon.

`int I.isSingleCrisp()`
: returns true if the polygon is a non empty convex crisp interval.

`bool I.isConvex()`
: returns true if the polygon is convex.

`bool I.isMonotone()`
: returns true if the polygon is monotone. (Def. 2.3.54)

`bool I.isSymmetric()`
: returns true if the polygon is symmetric. (Def. 2.3.54)

`CX I.SymmetryAxis2()`
: returns twice the x-coordinate of the symmetry axis and throws an error if `I` is not symmetric.

`int I.index(CX x)`
: returns the index of the rightmost polygon point that is left of `x`, or -1 if there is no such point. (Def. 2.3.20)

`int I.indexMax(bool front)`
: if `front` $=$ *true* it returns the index of the leftmost point with maximal y-value, otherwise it returns the index of the rightmost point with maximal y-value. If the polygon is empty it returns -1. (Def. 2.3.20)

`CY I.inf()`
: returns the smallest y-value of the polygon. (Def. 2.3.27)

`CY I.sup()`
: returns the hight sup(`I`) of the polygon. (Def. 2.3.27)

`float I.member(CX x)`
: returns the membership value for the x-coordinate x. (Def. 2.3.22)

`CX I.size2I(int k, int l)`
: returns 2 * the area below the polygon from vertex k to vertex l. (Def. 2.3.28)

`CX I.size2()`
: returns 2 * the area below the polygon. (Def. 2.3.28)

`CX I.size2(CX a, CX b)`
: returns 2* the area below the polygon from x-coordinate `a` to x-coordinate `b`. (Def. 2.3.28)

`CX I.centrePoint(int k, int m)`
: returns the x-coordinates of the `k`, `m`-center point. (Def. 2.3.29)

`int I.nComponents()`
returns the number of components of the interval. (Def. 2.3.31)

`Interval I.component(int k)`
returns the $\texttt{k}^{th}$ component of `I`. It throws an error if $\texttt{k} < 0$. (Def. 2.3.32)

`Region` is an enumeration type with values `support`, `core`, `kernel`, `maximum`.

`bool I.nextComponent(CX& x1, CX& x2, int& i1, int& i2, Region region)`
enumerates the components of the corresponding region (Def. 2.3.34)

`CX I.size(Region r)`
returns the size of the core/support/kernel/maxRegion. (Def. 2.3.35)

`Interval I.crisp(Region r)`
returns the core/support/kernel/maxRegion as crisp interval. (Def. 2.3.36)

`CX I.side(Region r, bool front)`
returns the x-coordinate of the left/rightmost point of the core/support/kernel/maxRegion. (Def. 2.3.37)

Hull Calculations

`Interval I.crispHull()`
returns the crisp hull of `I`. (Def. 2.3.38)

`Interval I.monotoneHull()`
returns the monotone hull of `I`. (Def. 2.3.39)

`Interval I.convexHull()`
returns the convex hull of `I`. (Def. 2.3.40)

Point–Interval Relations for Regions (Sec. 2.3.2.3)

`bool I.before(CX t, Region region)`
returns true if `t` is before the corresponding region of `I`.

`bool I.starts(CX t, Region region)`
returns true if `t` equals the left endpoint of the corresponding region of `I`.

`bool I.during(CX t, Region region)`
returns true if `t` is in the corresponding region of `I`.

`bool I.finishes(CX t, Region region)`
returns true if `t` equals the right endpoint of the corresponding region of `I`.

`bool I.after(CX t, Region region)`
returns true if `t` is after the corresponding region of `I`.

`bool I.between(CX t, Region region)`
returns true if `t` is in a gap between the corresponding region of `I`.

Relations Between Crisp Intervals and Regions of Fuzzy Intervals (Sec. 2.3.2.3)

`bool I.before(CX t1, CX t2, Region region)`
returns true if `[t1,t2[` is before the corresponding region of `I`.

`bool I.meets(CX t1, CX t2, Region region)`
: returns true if `[t1,t2[` meets the corresponding region of `I`.

`bool I.overlaps(CX t1, CX t2, Region region)`
: returns true if `[t1,t2[` overlaps the corresponding region of `I`.

`bool I.starts(CX t1, CX t2, Region region)`
: returns true if `[t1,t2[` starts the corresponding region of `I`.

`bool I.during(CX t1, CX t2, Region region)`
: returns true if `[t1,t2[` is during the corresponding region of `I`.

`bool I.finishes(CX t1, CX t2, Region region)`
: returns true if `[t1,t2[` finishes the corresponding region of `I`.

`bool I.after(CX t1, CX t2, Region region)`
: returns true if `[t1,t2[` is after the corresponding region of `I`.

`bool I.between(CX t1, CX t2, Region region)`
: returns true if `[t1,t2[` is during a gap of the corresponding region of `I`.

`bool I.disjoint(CX t1, CX t2, Region region)`
: returns true if `[t1,t2[` is disjoint with the corresponding region of `I`.

`CX I.partInside(CX t1, CX t2, Region region)`
: returns the size of the part of `[t1,t2[` which is inside the corresponding region of `I`.

Relations Between the Regions of two Fuzzy Intervals (Sec. 2.3.2.3)

`bool I.before(Interval J, Region region)`
: returns true if the corresponding region of `I` is before the corresponding region of `J`.

`bool I.meets(Interval J, Region region)`
: returns true if the corresponding region of `I` meets the corresponding region of `J`.

`bool I.overlaps(Interval J, Region region)`
: returns true if the corresponding region of `I` overlaps with the corresponding region of `J`.

`bool I.starts(Interval J, Region region)`
: returns true if the corresponding region of `I` starts the corresponding region of `J`.

`bool I.during(Interval J, Region region)`
: returns true if the corresponding region of `I` is during the corresponding region of `J`.

`bool I.finishes(Interval J, Region region)`
: returns true if the corresponding region of `I` finishes the corresponding region of `J`.

`bool I.equals(Interval J, Region region)`
: returns true if the corresponding region of `I` equals the corresponding region of `J`.

`bool I.disjoint(Interval J, Region region)`
: returns true if the corresponding region of `I` is disjoint with the corresponding region of `J`.

`CX I.partInside(Interval J, Region region)`
: yields the size of the part of the corresponding region of `I` which is inside the corresponding region of `J`.

Basic Unary Transformations (Def. 2.2.30)

`Interval I.extend(true)`
returns the rising part of `I`. (Def. 2.3.41)

`Interval I.extend(false)`
returns the falling part of `I`. (Def. 2.3.41)

`Interval I.scaleUp()`
scales the y-values of the interval up to $\top$. (Def. 2.3.41)

`Interval I.scaleUpD()`
is the destructive version of ScaleUp.

`Interval I.shift(CX a)`
shifts the interval by a units. (Def. 2.3.41)

`Interval I.shiftD(CX a)`
is the destructive version of Shift.

`Interval I.cut(CX x1, CX x2)`
cuts the part of the interval between `x1` and `x2`. (Def. 2.3.42)

`Interval I.cutI(int i1, int i2)`
cuts the part of the interval between the points with index `i1` and `i2`. (Def. 2.3.42)

`Interval I.times(float a)`
multiplies the y-values of the interval by `a`. (Def. 2.3.43)

`Interval I.exponentiate(float e)`
exponentiates the y-values of the interval with `e`. (Def. 2.3.47)

`Interval I.integrate(true)`
computes $J(x) \stackrel{\text{def}}{=} \int_{-\infty}^{x} \mathtt{I}(y)dy/|\mathtt{I}|$. `I` may be infinite. (Def. 2.3.45)

`Interval I.integrate(false)`
computes $J(x) \stackrel{\text{def}}{=} \int_{x}^{+\infty} \mathtt{I}(y)dy/|\mathtt{I}|$. `I` may be infinite. (Def. 2.3.45)

`Interval I.negate()`
inverts the y-values. (Def. 2.2.30)

`Interval I.invert()`
inverts the y-values of the gaps in the interval. (Def. 2.2.30)

`CY I.integrateAsymmetric(Interval J)`
computes $\int \mathtt{I}(x) \cdot \mathtt{J}(x)\ dx/|\mathtt{I}|$. `I` and `J` may be infinite. (Def. 2.3.51)

`CY I.integrateSymmetric(Interval J,bool simple)`
computes $\int \mathtt{I}(x) \cdot \mathtt{J}(x)\ dx/N(\mathtt{I},\mathtt{J})$. It throws an error if both `I` and `J` are infinite. (Def. 2.3.53)

Fuzzification

`Interval I.fuzzifyLinear(bool front, CX x1, CX x2, CX offset)`
linear fuzzification of the front/end part of the interval with absolute coordinates. (Def. 2.2.33)

`Interval I.fuzzifyLinear(bool front, float percent, float offset)`
linear fuzzification of the front/end part of the interval with relative coordinates. (Def. 2.2.36)

`Interval I.fuzzifyLinear(float percent, float offset)`
linear fuzzification of both sides of the interval with relative coordinates.

`Interval I.fuzzifyGaussian(bool front, CX xh, CX x0, CX offset)`
: Gaussian fuzzification of the front/end part of the interval with absolute coordinates. (Def. 2.2.34)

`Interval I.fuzzifyGaussian(bool front, float percent, float offset)`
: Gaussian fuzzification of the front/end part of the interval with relative coordinates. (Def. 2.2.36)

`Interval I.fuzzifyGaussian(float percent, float offset)`
: Gaussian fuzzification of both sides of the interval with relative coordinates.

General Transformations

`Interval I.unaryTransformation(UnaryYFunction f)`
: applies the unary y-function `f` to `I`. (Def. 2.3.46)

`Interval I.binaryTransformation(Interval J, BinaryYFunction f)`
: applies the binary y-function `f` to `I` and `J`. (Def. 2.3.50)

A.1.3 Circular Intervals

The CircularInterval class is a subclass of the Interval class. It manages and manipulates circular fuzzy temporal intervals (Sec. 2.4). The only change to the Interval class concerns the constructor functions. They take as an extra parameter the `xLimit`. This is the largest admissible x-coordinate. If, for example, angles between 0 and 360 are to be represented then xLimit must be 360.

Constructors

`Interval(CX xLimit)`
: constructs an empty circular interval.

`Interval(Point p, CX xLimit)`
: constructs an interval with a single point `p`. The x-coordinate of the point is taken modulo `xLimit`

`Interval(CX x, CY y, CX xLimit)`
: constructs an interval with a single point (`x` *modulo*`xLimit`, `y`).

`Interval(CX a, CX b, CX xLimit)`
: constructs a circular crisp interval [`a` *modulo* `xLimit`, `b`′[. `b`′ may be shifted by `xLimit`. For example, $[350, 10[$ yields $[350, 370[$.

`Interval(vector<Point> points, CX xLimit)`
: constructs an interval with a vector of points. The x-coordinates of the points are normalized.

`Interval(string s, CX xLimit)`
: constructs an interval from a string representation $[x_1, y_1\ x_2, y_2\ ...[$. The x-coordinates of the points are normalized.

A.1.4 Y-Functions

The unary and binary transformation methods (Def. 2.3.46, 2.3.50) expect a function f which is to be applied to one or two y-coordinates. Some of these functions, however, depend on extra parameters. For example the λ-complement (Def. 2.2.18) $n_\lambda(y) \overset{\text{def}}{=} \frac{1-y}{1+\lambda y}$ depends on the parameter λ. This would not be a problem in most functional programming languages. One can define $n(\lambda, x)$ and then get n_λ through currying. The solution in object oriented languages is a bit different. One defines a class “lambdaComplement” with instance variable “lambda”. The class can be instantiated with a corresponding value for lambda. This instance can now be used like any other data object in the language. The trick which allows one to use the instance like a function depends on the programming language. In C++ one can define a () operator for this class, which realizes the function application. If the instance is bound to the variable f, and x is another variable then $f(x)$ is now a legal expression and yields the function value. In Java one would define an apply-method and write $f.apply(x)$. The class-approach has many advantages: the parameters can be changed at any time, which is not so easy for curried functions; a class hierarchy can structure the functions according to their semantics, and not their types; further methods can be defined which do other kinds of computations and return meta-information, for example whether the function is linear.

FuTI realizes y-functions with the class hierarchy in Fig. A.1.

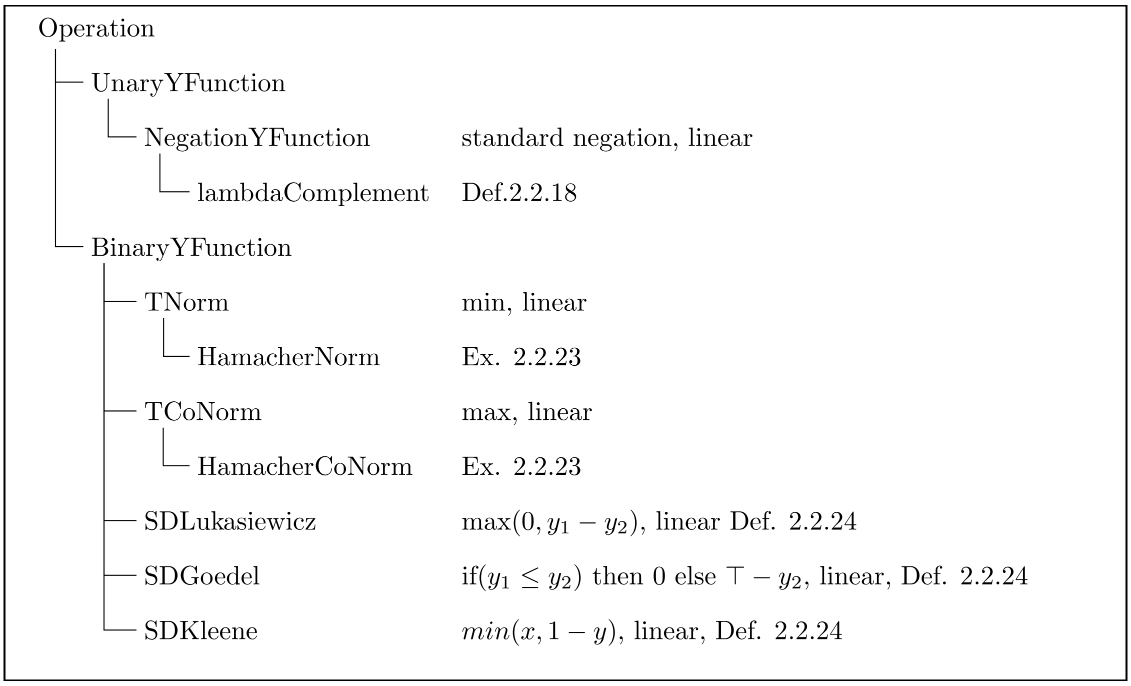


Figure A.1: Class Hierarchy for Y-Functions

The top class, ‘Operation’, manages the mapping of function names to the functions (instances of the other classes). Each instance can get a name, for example ‘myFavoriteLambdaComplement’, and one can retrieve the corresponding instance with the method `Operation::getOperation(string name)`. The name is optional. Instances without names are not accessible via `getOperation`.

Constructors

`NegationYFunction(string name)`
: constructs the standard negation function $\lambda(y)(1-y)$. (Def. 2.2.18)

`lambdaComplement(float lambda, string name)`
: constructs the lambda complement $\lambda(y)\frac{1-y}{1+\texttt{lambda}\ y}$. (Def. 2.2.18)

`TNorm(string name)`
: Constructs the min t-norm.

`HamacherNorm(float gamma, string name)`
: Constructs the Hamacher t-norm $\lambda(x,y)\frac{xy}{\texttt{gamma}+(1-\texttt{gamma})(x+y-xy)}$. (Def. 2.2.23)

`TCoNorm(string name)`
: Constructs the max t-conorm

`HamacherCoNorm(float beta, string name)`
: Constructs the Hamacher t-conorm $\lambda(x,y)\frac{x+y+(\texttt{beta}-1)xy}{1+\texttt{beta}\ xy}$. (Def. 2.2.23)

Parameter Modification

The parameters `lambda`, `gamma`, `beta` can be changed with `setParameter` and read with `getParameter`.

Appendix B

The GeTS–Language

B.1 Overview over the Language Constructs

The language constructs are summarized and briefly explained.

B.1.1 Types

Data Structure Types

`Integer`	standard integers
`Time`	very long integers
$\mathbb{F}$loat	standard floating point numbers
`String`	strings
`Interval`	fuzzy intervals
`Partitioning`	partitionings
`Label`	labels for partitions
`Duration`	durations
`DateFormat`	date formats

Enumeration Types

type name	possible values
`Bool`	`true/false`
`Side`	`left/right`
`PosNeg`	`positive/negative`
`UpDown`	`up/down`
`IntvRegion`	`core/kernel/support`
`PointRegion`	`core/kernel/support/maximum`
`Hull`	`core/kernel/support/crisp/monotone/convex`
`Fuzzify`	`linear/gaussian`
`Inclusion`	`subset/overlaps/bigger_part_inside`
`SplitInclusion`	`align/subset/overlaps/bigger_part_inside`
`Sequencing`	`sequential/overlapping/with_gaps`
`SDVersion`	`Kleene/Lukasiewicz/Goedel`

B.1.2 Arithmetics

Binary Arithmetic Operators
The operators are + (addition), - (subtraction), * (multiplication), (division), % (modulo), `max`, `min`, `pow` (exponentiation) (Def. 4.1.14).

Unary Arithmetic Operators
- (negation), `float`(b) (`Bool` $\mapsto \mathbb{F}$loat), `round`(a), `round`(a,`up/down`) (Def. 4.1.15).

Comparisons
<, <=, >, >= (Def. 4.1.16).
==, != (Def. 4.1.17).

B.1.3 Boolean Operators

- (complement), `and` or ‘&&’ (conjunction), `or` or ‘||’ (disjunction), `xor` or ‘ˆ’ (exclusive or) (Def. 4.1.18).

B.1.4 Control Constructs

`if` c `then` a `else` b (Def. 4.1.19).

`case` $C_1 : E_1, ..., C_n : E_n$ `else` D (Def. 4.1.20).

`while` c $\{E_1, ..., E_n\}$ D (Def. 4.1.21).

`Let` *variable* $=$ *expression*1 `in` *expression*2 (local binding) (Def. 4.1.22).

`dLet` $year|month|... =$ `date`$(time, dateFormat)$ `in` *expression* (local binding of dates) (Def. 4.1.65).

$x := E$ (assignment) (Def. 4.1.23).

B.1.5 Time Points

`now()` of type `Time` (current moment in time) (Def. 4.1.25)

`shift`$(time, duration, asGranule, dateOriented)$ of type `Time`$*$`Duration`$*$`Bool`$*$`Bool` $\mapsto$ `Time` (time shift by a duration) (Def. 4.1.28)

`shiftLength`$(time, duration, asGranule, dateOriented)$ of type `Time`$*$`Duration`$*$`Bool`$*$`Bool` $\mapsto$ `Time` (length of a time shift by a duration) (Def. 4.1.28)

`isInfinity(time)`	`Time` $\mapsto$ `Bool`
`isInfinity(time,positive/negative)`	`Time` $*$ `PosNeg` $\mapsto$ `Bool` (Def. 4.1.36)

`length`$(t_1, t_2, partitioning, asGranule)$ of type `Time` $*$ `Time` $*$ `Partitioning` $*$ `Bool` $\mapsto$ $\mathbb{F}$loat (length between t_1 ,t_2 in terms of the partitioning or granule) (Def. 4.1.42)

B.1.6 Intervals

[] of type `Interval` (empty interval)

[t1,t2] of type `Interval` (new crisp interval from t1 until t2)

`pushback`$(I, time, value)$ of type `Interval` $*$ `Time` $*$ $\mathbb{F}$loat $\mapsto$ `Void` adds $(time, value)$ to the membership function of the interval (Def. 4.1.30).

Set Operations on Intervals

`complement`(I)
  of type `Interval` $\mapsto$ `Interval`
`complement`(I, λ)
  of type `Interval` $*$ $\mathbb{F}$loat $\mapsto$ `Interval`
`complement`$(I, negation\_function)$
  of type `Interval` $*$ ($\mathbb{F}$loat $\mapsto$ $\mathbb{F}$loat) $\mapsto$ `Interval` (Def. 4.1.31)

`union`(I, J)
  of type `Interval` $*$ `Interval` $\mapsto$ `Interval`
`union`(I, J, β)
  of type `Interval` $*$ `Interval` $*$ $\mathbb{F}$loat $\mapsto$ `Interval`
`union`(I, J, co_norm)
  of type *Interval* $*$ `Interval` $*$ ($\mathbb{F}$loat $*$ $\mathbb{F}$loat $\mapsto$ $\mathbb{F}$loat) $\mapsto$ `Interval` (Def. 4.1.32)

$\texttt{intersection}(I, J)$
of type $\texttt{Interval} * \texttt{Interval} \mapsto \texttt{Interval}$
$\texttt{intersection}(I, J, \gamma)$
of type $\texttt{Interval} * \texttt{Interval} * \mathbb{F}\text{loat} \mapsto \texttt{Interval}$
$\texttt{intersection}(I, J, norm))$
of type $\texttt{Interval} * \texttt{Interval} * (\mathbb{F}\text{loat} * \mathbb{F}\text{loat} \mapsto \mathbb{F}\text{loat}) \mapsto \texttt{Interval}$ (Def. 4.1.33)

$\texttt{setdifference}(I, J)$
of type $\texttt{Interval} * \texttt{Interval} \mapsto \texttt{Interval}$
$\texttt{setdifference}(I, J, version)$
of type $\texttt{Interval} * \texttt{Interval} * \texttt{SDVersion} \mapsto \texttt{Interval}$
$\texttt{setdifference}(I, J, intersection, complement)$
of type $\texttt{Interval} * \texttt{Interval} * (\texttt{Interval} * \texttt{Interval} \mapsto \texttt{Interval}) *$
$(\texttt{Interval} \mapsto \texttt{Interval}) \mapsto \texttt{Interval}$ (Def. 4.1.34)

Predicates on Intervals

$\texttt{isCrisp}(I)$	$\texttt{Interval} \mapsto \texttt{Bool}$
$\texttt{isCrisp}(I, \texttt{left/right})$	$\texttt{Interval} * \texttt{Side} \mapsto \texttt{Bool}$
`isEmpty(I)`	$\texttt{Interval} \mapsto \texttt{Bool}$
`isConvex(I)`	$\texttt{Interval} \mapsto \texttt{Bool}$
`isMonotone(I)`	$\texttt{Interval} \mapsto \texttt{Bool}$
`isInfinite(I)`	$\texttt{Interval} \mapsto \texttt{Bool}$
`isInfinite(I,left/right)`	$\texttt{Interval} * \texttt{Side} \mapsto \texttt{Bool}$ (Def. 4.1.35)

$\texttt{during}(time, I, \texttt{core/kernel/support})$ of type $\texttt{Time} * \texttt{Interval} * \texttt{IntvRegion} \mapsto \texttt{Bool}$ checks whether $time$ is in the corresponding region of the interval I (Def. 4.1.37).

$\texttt{isSubset}(I, J, \texttt{core/kernel/support})$ of type $\texttt{Interval} * \texttt{Interval} * \texttt{IntvRegion} \mapsto \texttt{Bool}$ checks whether the corresponding region of I is a subset of the corresponding region of J (Def. 4.1.37).

$\texttt{doesOverlap}(I, J, \texttt{core/kernel/support})$ of type $\texttt{Interval} * \texttt{Interval} * \texttt{IntvRegion} \mapsto \texttt{Bool}$ checks whether the corresponding region of I overlaps with the corresponding region of J (Def. 4.1.37).

$\texttt{member}(time, I)$ of type $\texttt{Time} * \texttt{Interval} \mapsto \mathbb{F}\text{loat}$ (membership function) (Def. 4.1.38).

$\texttt{components}(I)$ of type $\texttt{Interval} \mapsto \texttt{Integer}$ (number of components of I) (Def. 2.2.10).

$\texttt{component}(I, k)$ of type $\texttt{Interval} * \texttt{Integer} \mapsto \texttt{Interval}$ (kth component of I) (Def. 2.2.10).

$\texttt{size}(I)$ of type $\texttt{Interval} \mapsto \texttt{Time}$ (size of the interval) (Def. 4.1.40)

$\texttt{size}(I, region)$ of type $\texttt{Interval} * \texttt{IntvRegion} \mapsto \texttt{Time}$ (size of the corresponding region of the interval) (Def. 4.1.40)

$\texttt{size}(I, t_1, t_2)$ of type $\texttt{Interval} * \texttt{Time} * \texttt{Time} \mapsto \texttt{Time}$ (size of the interval between t_1 and t_2) (Def. 4.1.40)

$\texttt{sup}(I)$ of type $\texttt{Interval} \mapsto \mathbb{F}\text{loat}$ (supremum of I) (Def. 4.1.41)

$\texttt{inf}(I)$ of type $\texttt{Interval} \mapsto \mathbb{F}\text{loat}$ (infimum of I) (Def. 4.1.41)

$\texttt{point}(I, side, region)$ of type $\texttt{Interval} * \texttt{Side} * \texttt{PointRegion} \mapsto \texttt{Time}$ (position of the corresponding end of the region) (Def. 4.1.44).

$\texttt{centerPoint}(I, n, m)$ of type $\texttt{Interval} * \texttt{Integer} * \texttt{Integer} \mapsto \texttt{Time}$ (n-m center point) (Def. 4.1.46).

Manipulation of Intervals
`shift`(I, t) of type `Interval` $*$ `Time` $\mapsto$ `Interval` shifts the interval by the given time (Def. 4.1.47).

`cut`(I, t_1, t_2) of type `Interval` $*$ `Time` $*$ `Time` $\mapsto$ `Interval` (extracts the part of I between t_1 and t_2) (Def. 4.1.48).

`hull`$(I,$ `core/support/kernel/crisp/monotone/convex`$)$ of type `Interval`$*$`Hull` $\mapsto$ `Interval` (construction of the corresponding hull) (Def. 4.1.49).

`invert`(I) of type `Interval` $\mapsto$ `Interval` inverts the membership function (Def. 4.1.50).

`scaleup`(I) of type `Interval` $\mapsto$ `Interval` scales the membership function up to maximal value 1 (Def. 4.1.51).

`times`(I, f) of type `Interval` $*$ $\mathbb{F}$loat $\mapsto$ `Interval` multiplies the membership function of I with f (Def. 4.1.52).

`exp`(I, e) of type `Interval` $*$ $\mathbb{F}$loat $\mapsto$ `Interval` exponentiates the membership function of I with e (Def. 4.1.52).

`extend`$(I,$ `positive/negative`$)$ of type `Interval`$*$`PosNeg` $\mapsto$ `Interval` extends I to the infinity (Def. 4.1.53).

`extend`$(I, \textit{length}, \textit{side})$ of type `Interval` $*$ `Time` $*$ `Side` $\mapsto$ `Interval` extends or shrinks I (Def. 4.1.54).

`integrate`$(I,$ `positive/negative`$)$ of type `Interval`$*$`PosNeg` $\mapsto$ `Interval` integrates the membership function (Def. 4.1.55).

`fuzzify`$(I,$ `linear/gaussian,left/right`$, \textit{increase}, \textit{offset})$
of type `Interval`, `Fuzzify`, `Side`, $\mathbb{F}$loat, $\mathbb{F}$loat $\mapsto$ `Interval`
`fuzzify`$(I,$ `linear/gaussian,left/right`$, x1, x2, \textit{offset})$
of type `Interval`, `Fuzzify`, `Side`, `Time`, `Time`, `Time` $\mapsto$ `Interval` (Def. 4.1.57)
fuzzifies the interval at the given side with the given fuzzification function.

`integrateSymmetric`$(I, J, \textit{simple})$
of type`Interval` $*$ `Interval` $*$ `Bool` $\mapsto$ $\mathbb{F}$loat and
`integrateAsymmetric`(I, J)
of type `Interval` $*$ `Interval` $\mapsto$ $\mathbb{F}$loat
symmetric or asymmetric integration of the membership function of I over the membership function of J (Def. 4.1.59).

`MaximizeOverlap`(I, J, EJ, D) of type `Interval`$*$`Interval`$*$`Interval`$*$(`Interval`$*$`Interval` $\mapsto$ $\mathbb{F}$loat) $\mapsto$ $\mathbb{F}$loat (Def. 4.1.61)

B.1.7 Time and Partitions

`time`$(\textit{year}|\textit{month}|...,\textit{dateFormat})$ of type `Integer`$|\dots|$`Integer` $*$ `DateFormat` $\mapsto$ `Time` maps a date in a given date format to the time point denoted by this date (Def. 4.1.64).

`partition`$(\textit{time}, \textit{partitioning})$ of type `Time` $*$ `Partitioning` $\mapsto$ `Interval`
`partition`$(\textit{time}, \textit{partitioning}, n, m)$ of type
`Time` $*$ `Partitioning` $*$ `Integer` $*$ `Integer` $\mapsto$ `Interval`
compute partitions as intervals (Def. 4.1.66).

`partitionBoundary`$(\textit{time}, \textit{partitioning},$ `left/right`$)$ of type `Time` $*$ `Partitioning` $*$ `Side` $\mapsto$ `Time` compute partition boundaries (Def. 4.1.67).

`partitioningIsBounded`$(P,$ `left/right`$)$ of type `Partitioning`$*$`Side` $\mapsto$ `Bool` checks whether the valid region of the partitioning is bounded (Def. 4.1.68).

`partitioningBoundary`$(P,$ `left/right`$)$ of type `Partitioning` $*$ `Side` $\mapsto$ `Time` returns the boundaries of the valid region of the partitioning (Def. 4.1.68).

`which`$(time, P, Q, inclusion, asGranule)$ of type `Time`$*$`Partitioning`$*$`Partitioning`$*$`Inclusion`$*$ `Bool` $\mapsto$ `Time` (which week in the year, for example) (Def. 4.1.69)

Labels
`label`$(time, partitioning)$ of type `Time` $*$ `Partitioning` $\mapsto$ `Label` returns the label of the corresponding partition (Def. 4.1.70).

`isLabel`$(label)$ of type `Label` $\mapsto$ `Bool` checks whether the label is not the NULL label (Def. 4.1.70).

`isGap`$(label)$ of type `Label` $\mapsto$ `Bool` checks whether the label is the gap label (Def. 4.1.70).

`LabelName`$(string)$ of type `String` $\mapsto$ `Bool` turns the string into a label (Def. 4.1.70).

`extractLabelled`$(I, label, partitioning, inclusion, intersect)$ of type `Interval`$*$`Label`$*$`Partitioning`$*$ `SplitInclusion` $*$ `Bool` $\mapsto$ `Interval` extracts partitions in the interval I with the given label (Def. 4.1.71).

`nextGranule`$(time, partitioning, label, n, withGaps)$ of type `Time` $*$ `Partitioning` $*$ `Label` $*$ `Integer` $*$ `Bool` $\mapsto$ `Interval` constructs a new interval which represents a granule (Def. 4.1.72).

Loops over Intervals
`componentwise`$(I, initialObject, operation, combination)$ of type `Interval` $*\, T *$ (`Interval` $\mapsto$ $T) * (T * T \mapsto T) \mapsto T$ applies *operation* to all components of the interval (Def. 4.1.73).

`split`$(I, duration, asGranule, dateOriented, initialObject, operation,$
$combination, region, forward, inclusion, sequencing, intersect)$ of type
`Interval` $*$ `Duration` $*$ `Bool` $*$ `Bool` $*\, T *$ (`Interval` $\mapsto T) *$ (`Interval` $*$ `Interval` $\mapsto T) *$ `IntvRegion` $*$ `Bool` $*$ `SplitInclusion` $*$ `Sequencing` $*$ `Bool` $\mapsto T$
splits the interval into parts and applies the operation to them (Def. 4.1.74).

B.2 The Application Programming Interface

The C++ API of the GeTS language is as follows:

GeTS functions are realized as a class `Function` in a namespace `GeTS`. They can be defined, they can be applied to arguments, and some information about them can be retrieved.

Definition:
A new GeTS function can be created with an ordinary constructor:

`fct = new Function(definition)`.

The `definition` is a string representation of the definition, optionally followed by the keyword `explanation` and some text. The explanation can be retrieved just by `fct->explanation`.

The definition is parsed and compiled. Parsing or compilation errors can be obtained by `fct->getError()`. The function `fct->noError()` checks whether there was a parsing or compilation error.

Information about Functions:
The function definitions can be obtained in different versions:

`fct->callString()` returns the function call as string

`fct->typeString()` returns the function type as string

`fct->definitionString()` returns the function definition with line numbering as string.

`fct->codeString()` returns the abstract machine code as string.

Example B.2.1 (for `codeString()`) The code string for the function

```
PIRBefore(Time t, Interval I) =
     if (isEmpty(I) or isInfinite(I,left)) then false
     else (t < point(I,left,support))
```

(Example 4.1.3) is

```
0: I[1,Interval]
1: isEmpty(Interval->Bool)
2: ||(Bool*Bool->Bool)
3: I[1,Interval]
4: left[-1,left,Side]
5: isInfinite(Interval*Side->Bool)
6: ||(Bool*Bool->Bool)
7: IfThenElse(Bool*Bool*Bool->Bool)
8: false[-1,false,Bool]
9: IfThenElse(Bool*Bool*Bool->Bool)
10: t[0,Time]
11: I[1,Interval]
12: left[-1,left,Side]
13: support[-1,support,IntvRegion]
14: point(Interval*Side*IntvRegion->Time)
15: <(Time*Time->Bool)
16: IfThenElse(Bool*Bool*Bool->Bool)
```

It should be fairly obvious what this means. For example, line 0, `I[1,Interval]` means that the parameter `I` at parameter position 1 and of type `Interval` is pushed to the stack. Line 1: `isEmpty(Interval->Bool)` means that the `isEmpty` predicate pops its argument from the stack, performs the check, and pushes the result to the stack again. Line 2: `||(Bool*Bool->Bool)` is the first invocation of the `or` check. It checks the top of stack. If this is the Boolean value `true` then this value is popped from the stack and the program counter is set to 7. The remaining program steps are more or less self explaining. ∎

It should be noticed that the actual computations, for example, integrating over a membership function of an interval, are done with compiled machine code. The commands of the GeTS abstract machine are only used to control the invocation of this machine code.

Auxiliary Classes and Types:

The data types of GeTS are represented as a class `Type` in the namespace `GeTS`. They can be basic data types or compound types. The most important API method for types is `toString`. Most other methods are for internal use.

The data which are manipulated by a GeTS function are comprised into a union type. Without further explanation we just list the definition.

```
union GeTSValue {
  long long int*         Time;
  PartLib::Partitioning* Partitioning;
  PartLib::Label*        Label;
  PartLib::DateFormat*   DateFormat;
  FuTIRe::Interval*      Interval;
  Function*              lFunction;
  int                    Integer;
  float                  Float;
  bool                   Bool;
  DurationSpec*          Duration;
  string*                String;
};
```

Application:
There are two application functions:
`pair<Type*, GeTSValue> apply(vector<pair<Type*, GeTSValue> >& values)`
can be used to apply the function to a vector of parameters. The result is a pair consisting of the result type and the result value.

The other method

```
pair<Type*, GeTSValue>
   apply(const string& arguments,const vector<FuTIRe::Interval*>& intervals)
```

can be used to apply the function to a string representation of the parameters. Intervals are represented as non-negative integers. The integers are used as indices in the given vector of interval pointers. The result is again a type-value pair.

Bibliography

[1] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[2] T. Berners-Lee, M. Fischetti, and M. Dertouzos. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web.* Harper, San Francisco, September 1999. ISBN: 0062515861.

[3] C. Bettini and R.D.Sibi. Symbolic representation of user-defined time granularities. *Annals of Mathematics and Artificial Intelligence*, 30:53–92, 2000. Kluwer Academic Publishers.

[4] Claudio Bettini, Curtis E. Dyreson, William S. Evans, Richard T. Snodgrass, and X. Sean Wang. *Temporal Databases, Rreseach and Practice*, volume 1399 of *LNCS*, chapter A Glossary of Time Granularity Concepts, pages 406–413. Springer Verlag, 1998.

[5] Claudio Bettini, Sushil Jajodia, and Sean X. Wang. *Time Granularities in Databases, Data Mining and Temporal Reasoning.* Springer Verlag, 2000.

[6] Claudio Bettini, Sergio Mascetti, and X. Sean Wang. Mapping calendar expressions into periodical granularities. In C. Combi and G. Ligozat, editors, *Proc. of the 11th International Symposium on Temporal Representation and Reasoning*, pages 87–95, Los Alamitos, California, 2004. IEEE.

[7] François Bry, Frank-André Rieß, and Stephanie Spranger. CaTTS: Calendar Types and Constraints for Web Applications. research report, PMS-FB-2004-24 PMS-FB-2004-24, Institute for Informatics, University of Munich, 2004.

[8] François Bry, Frank-André Rieß, and Stephanie Spranger. A Reasoner for Calendric and Temporal Data. Forschungsbericht/research report PMS-FB-2005-18, Institute for Informatics, University of Munich, 2005.

[9] François Bry, Bernhard Lorenz, Hans Jürgen Ohlbach, and Stephanie Spranger. On reasoning on time and location on the web. In N. Henze F. Bry and J. Malusyński, editors, *Principles and Practice of Semantic Web Reasoning*, volume 2901 of *LNCS*, pages 69–83. Springer Verlag, 2003.

[10] Diana R. Cukierman. *A Formalization of Structured Temporal Objects and Repetition.* PhD thesis, Simon Franser University, Vancouver, Canada, 2003.

[11] Diana R. Cukierman and James P. Delgrande. Expressing time intervals and repetition within a formalization of calendars. *Computational Intelligence*, 14(4):563–597, 1998.

[12] Diana R. Cukierman and James P. Delgrande. The SOL time theory: A formalization of structured temporal objects and repetition. In C. Combi and G. Ligozat, editors, *Proc. of the 11th International Symposium on Temporal Representation and Reasoning*, pages 71–34, Los Alamitos, California, 2004. IEEE.

[13] Nachum Dershowitz and Edward M. Reingold. *Calendrical Calculations.* Cambridge University Press, 1997.

[14] Didier Dubois and Henri Prade, editors. *Fundamentals of Fuzzy Sets.* Kluwer Academic Publisher, 2000.

[15] Curtis E. Dyreson, Wikkima S. Evans, Hing Lin, and Richard T. Snodgrass. Efficiently supporting temporal granularities. *IEEE Transactions on Knowledge and Data Engineering*, 12(4):568–587, 2000.

[16] Lavinia Egidi and Paolo Terenziani. A lattice of classes of user-defined symbolic periodicities. In C. Combi and G. Ligozat, editors, *Proc. of the 11th International Symposium on Temporal Representation and Reasoning*, pages 13–20, Los Alamitos, California, 2004. IEEE.

[17] D. M. Gabbay, I. Hodkinson, and M Reynolds, editors. *Temporal Logic: Mathematical Foundations and Computational Aspects*, volume 1. Oxford: Clarendon Press, 1994.

[18] L. Godo and L. Vila. Possibilistic temporal reasoning based on fuzzy temporal constraints. In Chris S. Mellish, editor, *IJCAI'95: Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, volume 2, pages 1916–1922. IJCAI, 1995.

[19] Jacob E. Goodman and Joseph O'Rourke, editors. *Handbook of Discrete and Computational Geometry.* CRC Press, 1997.

[20] I.A. Goralwalla, Y. Leontiev, M.T. Ozsu, D. Szafron, and C. Combi. Temporal granularity: Completing the picture. *Journal of Intelligent Information Systems*, 16(1):41–63, 2001.

[21] Nick Kline, Jie Li, and Richard Snodgrass. Specifying multiple calendars, calendric systems and field tables and functions in timeadt. Technical Report TR-41, Time Center Report, May 1999.

[22] B. Leban, D. Mcdonald, and D.Foster. A representation for collections of temporal intervals. In *Proc. of the American National Conference on Artificial Intelligence (AAAI)*, pages 367–371. Morgan Kaufmann, Los Altos, CA, 1986.

[23] I. Navarette M.A. Cardenas and R. Marin. Efficient resolution mechanism for fuzzy temporal constraint logic. In *TIME'2000: Proc. of the Seventh International Workshop on Temporal Representation and. Reasoning*, pages 39–46. IEEE Press, 2000.

[24] Roque Marín, M. A. Cárdenas Viedma, M. Balsa, and J. L. Sanchez. Obtaining solutions in fuzzy constraint networks. *Int. J. Approx. Reasoning*, 16(3-4):261–288, 1997.

[25] Gábor Nagypál and Boris Motik. A fuzzy model for representing uncertain, subjective and vague temporal knowledge in ontologies. In *Proceedings of the International Conference on Ontologies, Databases and Applications of Semantics, (ODBASE)*, volume 2888 of *LNCS*. Springer-Verlag, 2003.

[26] M. Niezette and J. Stevenne. An efficient symbolic representation of periodic time. In *Proc. of the first International Conference on Information and Knowledge Management*, volume 752 of *Lecture Notes in Computer Science*, pages 161–169. Springer Verlag, 1993.

[27] Peng Ning, X. Sean Wang, and Sushil Jajodia. An algebraic representation of calendars. *Annals of Mathematics and Artificial Intelligenc*, 36(1-2):5–38, September 2002. Kluwer Academic Publishers.

[28] Hans Jüergen Ohlbach. Computational treatement of temporal notions – the CTTN system. In François Fages, editor, *Proceedings of PPSWR 2005*, Lecture Notes in Computer Science, pages 137–150, 2005. see also URL: http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-30.

[29] Hans Jürgen Ohlbach. About real time, calendar systems and temporal notions. In H. Barringer and D. Gabbay, editors, *Advances in Temporal Logic*, pages 319–338. Kluwer Academic Publishers, 2000.

[30] Hans Jürgen Ohlbach. Calendar logic. In I. Hodkinson D.M. Gabbay and M. Reynolds, editors, *Temporal Logic: Mathematical Foundations and Computational Aspec ts*, pages 489–586. Oxford University Press, 2000.

[31] Hans Jürgen Ohlbach. Fuzzy time intervals – the FuTI-library. Research Report PMS-FB-2005-26, Inst. für Informatik, LFE PMS, University of Munich, June 2005. URL: http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-26.

[32] Hans Jürgen Ohlbach. GeTS – a specification language for geo-temporal notions. Research Report PMS-FB-2005-29, Inst. für Informatik, LFE PMS, University of Munich, June 2005. URL: http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-29.

[33] Hans Jürgen Ohlbach. Modelling periodic temporal notions by labelled partitionings – the PartLib library. In S. Artemov, H. Barringer, A. d'Avila Garces, L. C. Lamb, and J. Woods, editors, *Essays in Honour of Dov Gabbay*, volume 2, pages 453–498. College Publications, King's College, London, 2005. ISBN 1-904987-12-5. See also http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-28.

[34] Hans Jürgen Ohlbach. Relations between fuzzy time intervals. Research Report PMS-FB-2005-27, Inst. für Informatik, LFE PMS, University of Munich, June 2005. URL: http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-27.

[35] Hans Jürgen Ohlbach. Periodic temporal notions as 'tree partitionings'. Forschungsbericht/research report PMS-FB-2006-11, Institute for Informatics, University of Munich, 2006.

[36] Joseph O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1998.

[37] Klaus U. Schulz and Felix Weigel. Systematics and architecture for a resource representing knowledge abo ut named entities. In Jan Maluszynski Francois Bry, Nicola Henze, editor, *Principles and Practice of Semantic Web Reasoning*, pages 189–208, Berlin, 2003. Springer-Verlag.

[38] Michael D. Soo and Richard T. Snodgrass. Mixed calendar query language support for temporal constants. Technical Report TR 92-07, Dept. of Computer Science, Univ. of Arizona, February 1992.

[39] L. Vila and L. Godo. On fuzzy temporal constraint networks. *Mathware and Soft Computing*, 3:315–334, 1994.

[40] L. Vila and L. Godo. Query-answering in fuzzy temporal constraint networks. In Chris S. Mellish, editor, *FUZZ-IEEE'95: IEEE International Conference on Fuzzy Systems Yokohama*, volume 1, pages 43–48, 1995.

[41] L. A. Zadeh. Fuzzy sets. *Information & Control*, 8:338–353, 1965.