# A1-D1

# Geotemporal Reasoning: Basic Theory

| | |
|---|---|
| Project number: | IST-2004-506779 |
| Project title: | Reasoning on the Web with Rules and Semantics |
| Project acronym: | REWERSE |
| Document type: | D (deliverable) |
| Nature of document | R (report) |
| Dissemination level: | PU (public) |
| Document number: | IST506779/Munich/A1-D1/D/PU/a1 |
| Responsible editor(s): | H.J. Ohlbach |
| Reviewer(s): | Claude Kirchner |
| Contributing participants: | Munich |
| Contributing workpackages: | A1 |
| Contractual date of delivery: | 31 August 2004 |

**Abstract**
This deliverable contains a detailed description of the bottom layers of the WebCal system for representing and manipulating 'geotemporal' information. The basic components are the FuTIRe library for representing fuzzy temporal intervals, the PartLib library for representing periodic temporal notions, and the mixed function layer, which deals with operations involving temporal intervals and periodic temporal notions.

A first approach for representing named entities is described in the last chapter of this deliverable. These ideas are the basis for a knowledge representation system with, so far, 2000 entries of named entities.

**Keyword List**
semantic web, temporal notions, named entities

# Geotemporal Reasoning: Basic Theory

**Hans Jürgen Ohlbach**[1], **Klaus Schulz**[2] **and Felix Weigel**[2]

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

[2] Centrum für Informations- und Sprachverarbeitung, University of Munich
Email: `schulz@cis.uni-muenchen.de`

[3] Centrum für Informations- und Sprachverarbeitung, University of Munich
Email: `weigel@cis.uni-muenchen.de`

1 September 2004

**Abstract**
This deliverable contains a detailed description of the bottom layers of the WebCal system for representing and manipulating 'geotemporal' information. The basic components are the FuTIRe library for representing fuzzy temporal intervals, the PartLib library for representing periodic temporal notions, and the mixed function layer, which deals with operations involving temporal intervals and periodic temporal notions.

A first approach for representing named entities is described in the last chapter of this deliverable. These ideas are the basis for a knowledge representation system with, so far, 2000 entries of named entities.

**Keyword List**
semantic web, temporal notions, named entities

# Contents

**1 Introduction** **1**
- 1.1 The WebCal System . . . 1
- 1.2 Named Entities . . . 4

**2 Fuzzy Time Intervals and Relations – The FuTIRe Library** **5**
- 2.1 Motivation and Introduction . . . 5
- 2.2 The Mathematics of Fuzzy Time Intervals and Relations . . . 6
  - 2.2.1 Fuzzy Time Intervals . . . 7
  - 2.2.2 Scalar Properties of Fuzzy Time Intervals . . . 10
  - 2.2.3 Functions operating on Fuzzy Time Intervals . . . 13
  - 2.2.4 Set Operators for Fuzzy Intervals . . . 15
  - 2.2.5 Hull Operators for Fuzzy Intervals . . . 20
  - 2.2.6 Basic Unary Transformations . . . 21
  - 2.2.7 Point–Interval Relations . . . 29
  - 2.2.8 Interval–Interval Relations . . . 36
  - 2.2.9 The Search Problem . . . 46
- 2.3 Nagypál and Motik's Interval–Interval Relations . . . 49
- 2.4 Datastructures and Algorithms . . . 54
  - 2.4.1 Points . . . 57
  - 2.4.2 Fuzzy Time Intervals . . . 61
- 2.5 The FuTIRe Interface . . . 77
  - 2.5.1 Points . . . 77
  - 2.5.2 Intervals . . . 78
  - 2.5.3 Y-Functions . . . 83
  - 2.5.4 Interval Operators . . . 84
  - 2.5.5 Specification of Unary Interval Operators . . . 89
  - 2.5.6 Interval–Interval Relations . . . 93
- 2.6 Summary . . . 96

**3 Modelling Periodic Temporal Notions by Labelled Partitionings of the Real Numbers – The PartLib Library** **97**
- 3.1 Motivation and Introduction . . . 97
- 3.2 Partitionings . . . 102
  - 3.2.1 Length of Intervals in Partitions . . . 104
  - 3.2.2 Labels . . . 106

3.2.3 Granules . . . . . 106
3.2.4 Relations Between Partitionings . . . . . 109
3.3 Date Formats . . . . . 109
3.4 Shift Functions . . . . . 111
3.4.1 Length Oriented Shift Function . . . . . 111
3.4.2 Date Oriented Shift Function . . . . . 112
3.4.3 The Shift Function for Granules . . . . . 113
3.5 Durations . . . . . 114
3.6 Global and Local Reference Time . . . . . 114
3.7 Specification of Partitionings . . . . . 115
3.7.1 Algorithmic Partitionings . . . . . 116
3.7.2 Duration Partitionings . . . . . 119
3.7.3 Regular Partitionings . . . . . 120
3.7.4 Date Partitionings . . . . . 122
3.7.5 Folded Partitionings . . . . . 123
3.8 Summary . . . . . 127
3.9 Appendix: The PartLib Interface . . . . . 127
3.9.1 Repetition Patterns . . . . . 127
3.9.2 The Label Class . . . . . 128
3.9.3 The Labelling Class . . . . . 129
3.9.4 The Duration Class . . . . . 131
3.9.5 Class DateFormat . . . . . 132
3.9.6 Class DateData . . . . . 133
3.9.7 The Partitioning Class . . . . . 133

**4 Calendrical Calculations with Time Partitionings and Fuzzy Time Intervals 139**
4.1 Motivation and Introduction . . . . . 139
4.2 Time Points and Time Intervals . . . . . 140
4.3 Partitionings . . . . . 142
4.4 Durations . . . . . 143
4.5 Operations on Points, Intervals and Partitionings . . . . . 143
4.5.1 Shift and Extend . . . . . 144
4.5.2 Which . . . . . 145
4.5.3 Extract . . . . . 146
4.6 Split . . . . . 149
4.7 Summary . . . . . 151

**5 Systematics and Architecture for a Resource Representing Knowledge about Named Entities 153**
5.1 Introduction . . . . . 153
5.2 Motivation: Goals, pictures, and intended applications . . . . . 155
5.3 Systematics and Organization . . . . . 158
5.3.1 Syntax of identifiers . . . . . 158
5.3.2 Meaning of identifiers . . . . . 159
5.3.3 Derived hierarchical structure . . . . . 161
5.3.4 Examples from the experimental version . . . . . 164
5.4 The structure of the entries . . . . . 164

5.5 Level 2: Facts . . . . . 167
5.6 Concluding remarks . . . . . 169

# Chapter 1

# Introduction

The work package A1 focuses on three different areas, which eventually will grow together to a powerful support system for understanding and manipulating geotemporal, geospacial and topical information. With geotemporal information we mean temporal notions from everyday life, and from web pages. This is in contrast to temporal logics, which deal with abstracted and simplified temporal phenomena, and which is mainly used for specifying and verifying state transition systems of any kind. Similarly, with geospacial information we mean information about locations and relations between locations, as we use them in every day life, and in Web pages. Both, geotemporal and geospacial notions may need to refer to named entities. 'during the reign of James II' is a temporal notions, which refers to the named entity 'James II'. 'at James II's birthplace' is a geospacial notion which refers to the same named entity 'James II'. Therefore, in order to understand such kinds of notions, we need a representation of named entities, or, more general, topics. Therefore this is the third area of research in A1.

In this first deliverable we address in depth a number of aspects of our approach to geotemporal information processing. This part of research is of course not yet finalized. Future deliverables will contain more material.

We also address in this deliverable a first approach for representing named entities. With the schema described in Chapter 5 we built up a database with 2000 entries. Currently we are revising the underlying knowledge representation scheme to make it more formal and to improve its expressiveness. The details will be presented in deliverable D5.

## 1.1 The WebCal System

WebCal is a computer program which provides advanced calendrical calculations for Web services. WebCal has an international dimension: it provides most of the calendar systems world-wide in use, with timezones, daylight savings time regulations, leap years, leap seconds etc. It also has an historical dimension: it models historical sequences of calendrical regulations, for example sequences of calendar systems, sequences of daylight savings time regulations, timezones changing over time etc. It has an application oriented dimension, i.e. one can define new application specific temporal notions, for example school holidays, financial years, ecclesiastical calendars, Mary's birthday, my own working hours, and a lot more. WebCal can deal with fuzzy notions like 'late night' or 'around noon', because it represents time intervals as fuzzy sets. The program is based on an algebraic model of basic temporal notions, which gives all the operations a very precise semantics. The main idea behind WebCal is to provide a few powerful data structures, and to offer applications as many operations as possible on these data structures.

A prototype of WebCal can be accessed via the Unix command 'telnet puka-puka.pms.ifi.lmu.de 1952'. The different parts of WebCal are currently being revised and documented. Figure 1.1 gives a rough overview of the system.

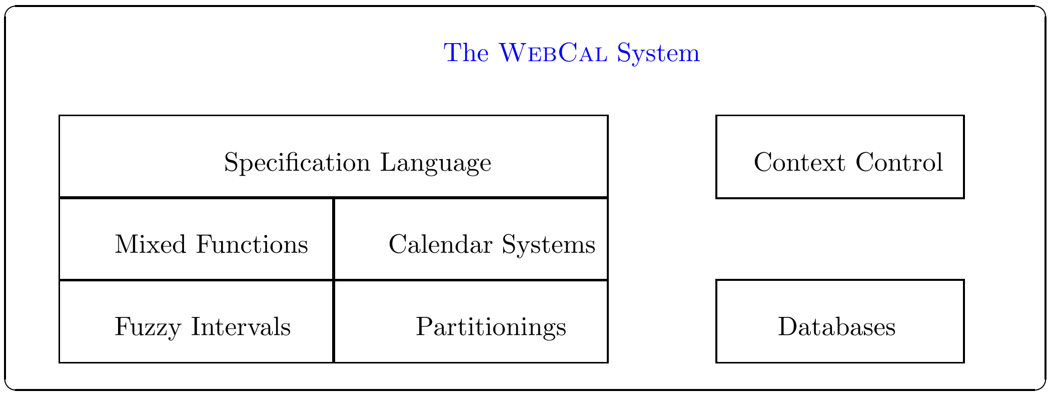


Figure 1.1: The WebCal System

The basic data structures in WebCal are time points, fuzzy time intervals, labelled partitionings for representing periodic temporal notions, calendars, and various kinds of operations, which can be combined in a user specified way.

Fuzzy time intervals are implemented in the FuTIRe library, which is described in Chapter 2. A short version of this part was published in [41]. Labelled partitionings are needed for representing all kinds of periodic temporal notions, from the basic time units, years, months etc., until user defined notions like 'my weekend'. They are implemented in the PartLib library, which is described in Chapter 3. A very short version of this chapter was published in [42].

A layer above the fuzzy intervals and the partitionings contains all the functions which involve both, an interval and a partitioning. The details are described in Chapter 4 (see also [39]).

Another part is the implementation of the calendar systems as collections of partitionings. The algorithms from Dershowitz and Reingold's book 'Calendrical Calculations' [16] are needed here. This part is implemented in the prototype, but not yet revised and documented.

The third layer contains a specification language for user defined temporal notions. A very first version of such a language was published in [37, 38, 44]. Most of the concrete details for the WebCal system are worked out already, but not yet documented.

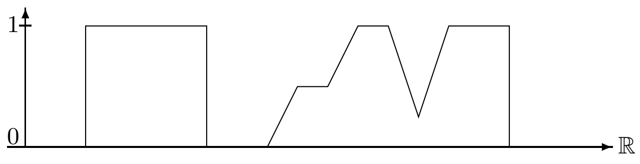
The 'Context Control' module is also yet to be developed. It will contain mappings from context information to control parameters for WebCal. An example is a mapping from countries to timezones. WebCal needs timezones as offset from UTC time. The context information may, however, be, for example: the timezones for Greece is needed. The actual timezones has now to be taken from a database.

Many parts of WebCal need access to databases. The mapping from countries to timezones is one example. Other examples are the daylight savings time regulations, the list of leap seconds, the use of calendar systems in the various countries, the dates of events to which one might refer (for example, 'after the Olympic Games in Athens'). All this is comprised in a database module, which is yet to be developed.

The Chapters 2 and 3 are the main parts of this deliverable. Chapter 2 is about fuzzy time intervals and Chapter 3 is about the representation of periodical temporal notions by means of labelled partitionings of the time axis. Since the chapters contain all the technical details, they are quite large. Therefore we give a very brief overview here.

## Fuzzy Temporal Intervals in the FuTIRe Library

The real numbers serve in FuTIRe as the time axis. A fuzzy time interval is therefore a fuzzy interval over the real numbers. Examples are:



*Crisp and Fuzzy Intervals*

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

FuTIRe contains a collection of classes and methods for measuring and manipulating these kinds of intervals. There are basically four different kinds of operations, which are available:

- basic construction and measurement functions for fuzzy time intervals;
- set operations like fuzzy union, intersection, set difference, which work with so-called T-norms and T-conorms;
- point-interval relations like before, starts, finishes, during, after, in_the_middle, in_the_gap etc. These relations are very natural for crisp intervals. Their definition is, however, not clear at all for proper fuzzy intervals. Therefore FuTIRe allows one to customize the relations by parameterizing them in different ways;
- interval-interval relations like before, meets, overlaps, starts during, finishes, after, until etc. In the crisp case these are Allen's standard interval relations. In the fuzzy case there are again different possibilities to define such relations. FuTIRe provides three schemes for the relations:
    1. Allen's relations between crisp approximations for the fuzzy sets;
    2. the interval-interval relations are generated from the corresponding point-interval relations by integrating the point-interval relations over one of the two intervals;
    3. Nagypál and Motik's scheme of fuzzy interval-interval relations, where the front and back parts of the fuzzy intervals are taken for the computation [33].

The fuzzy interval operations are not just functions or methods, which can be called in the usual way. FuTIRe provides them as first-class objects, which can be combined and manipulated in different ways. For example, one can specify new operations in a particular specification language by stating how the existing operations are to be composed to yield the new operation.

### Labelled Partitionings in the PartLib Library

Periodical temporal notions like years, months etc. are the building blocks of calendar systems. Other temporal notions where periodicity is an essential feature are timetables, school holidays, individual notions like 'my weekend' etc. Periodicity means that the time axis is partitioned into, usually finite, partitions. Quite often these partitions have names. For example, the months are named January, February etc. In the PartLib library one can specify partitionings of the time axis in different ways, and one can attach labels (names) to the partitions. A particular label is 'gap'. With 'gap', and the possibility to use the same label at different positions, one can specify groups of partitions with the same semantic meaning. In the literature they are usually called *granules*. For example, one can specify 'working hours' as the collection of partitions starting with a partition in the morning, labelled 'working hours', followed by a partition labelled 'gap' (the lunch break), followed bay another partition in the afternoon also labelled 'working hour', and then followed by a partition which is again labelled 'gap' (evening and night). The first three partitions make up the granule 'working hours'.

PartLib provides various functions for navigating through partitions and granules.

The partitionings can be specified in PartLib either algorithmically or symbolically. Algorithmic specifications make it easier to take into account irregular phenomena like leap seconds, daylight savings time schemes, leap years etc. Some partitionings can in fact only be specified algorithmically, for example the ecclesiastical calendar with the Easter date, which follows the moon cycle. Sunrise and sunset can also only be specified algorithmically. Many other periodic temporal notions are much easier specified symbolically. Examples are the seasons, school holidays etc. PartLib allows one to specify these notions by taking the relevant dates from a database. For the application interface, however, it makes no difference how the partitioning is defined.

Another important concept, which is implemented in PartLib, is the notion of *durations*. A duration is something like '3 months + 1 week'. It is a sequence of pairs (number, partitioning). Durations can be used to measure intervals (3.5 months long), or to specify time shifts (in 3.5 months).

Operations which involve partitionings or durations on one side, and intervals on the other side are implemented at the *mixed function layer*, which is a separate component of the WebCal system. A typical example is the shift operation for intervals (e.g. shift the holiday by one month and one week). Since the length of partitions (months) can vary, this is a nontrivial operation. Chapter 4 contains the details.

## 1.2 Named Entities

In Chapter 5 we describe the systematics and architecture of an experimental resource that contains a thematic-geographic-temporal hierarchy for classifying named entities of various kinds with respect to the hierarchy, lists synonyms, and gives formal descriptions of these entities and their relations. The resource should offer a general basis for semantic annotation, indexing, retrieval, querying, browsing and hyperlinking of (semi-)textual web documents, structured documents and flat texts. The text was published in [47].

The following four chapters are self contained papers. They describe systems which can also be used independently of WebCal.

# Chapter 2

# Fuzzy Time Intervals and Relations – The FuTIRe Library

**Hans Jürgen Ohlbach**

## Abstract

The FuTIRe library is a collection of classes and methods for representing and manipulating fuzzy time intervals and relations between them. Time intervals like 'tonight', which are usually not very precise, can be modeled as fuzzy sets. But this causes the problem that the relations between points and intervals and between two intervals, which are usually very trivial, become very complex when the intervals are fuzzy sets. Moreover, there are many different possibilities to define such relations. In FuTIRe it is not only possible to represent fuzzy time intervals, but one can define customized fuzzy point–interval and interval–interval relations. These relations can even yield fuzzy values when the intervals are in fact crisp. As an example for an application, consider a database with, say, a cinema timetable, and you query the timetable "give me all performances ending before midnight". The usual 'before' relation will exclude the performances ending a second after midnight. With the fuzzy before relation in FuTIRe you can get instead of a sharp drop to 0 at midnight decreasing fuzzy values after midnight, and these can be used to order the results of the query. FuTIRe is an open source C++ library.

## 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 chapter I describe the FuTIRe-system (Fuzzy Temporal Intervals and Relations), a library of datastructures and algorithms for representing and manipulating fuzzy temporal notions.

In FuTIRe one can

- represent finite and infinite fuzzy time intervals;
- combine and manipulate these intervals in various ways;
- define customized point–interval relations like ‘before’, ‘starts’, ‘during’ etc. between time points and crisp as well as fuzzy intervals in a fuzzy way;
- define customized interval–interval relations similar to Allen’s interval relations, but between fuzzy intervals, and with fuzzy values as result.

In the first part of the chapter I describe the components of the FuTIRe-system in a purely mathematical way, without any commitment to concrete datastructures and algorithms. In the second part I present a representation of the fuzzy intervals as polygons with integer coordinates. All algorithms in FuTIRe work on these polygons. Finally the concrete interface to the library is listed and explained. The library is a component of the WebCal-system [12], a program for evaluating temporal expressions like ‘three weeks after Easter’. WebCal is currently under development.

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

## 2.2 The Mathematics of Fuzzy Time Intervals and Relations

The mathematics of general fuzzy sets [53] has been investigated in great depth. The particular fuzzy sets in FuTIRe 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. [17]).

Since FuTIRe 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 organize them in a meaningful way and to motivate them with temporal notions and operations.

**Definition 2.2.1 (Basic Notions and Notations)** *We define some notions and notations about numbers and intervals.*

$\mathbb{N}$ *are the integers,* $\mathbb{R}$ *are the real numbers and* $\mathbb{R}^+ \overset{\text{def}}{=} \mathbb{R} \cup \{-\infty, +\infty\}$.
$\sup(s)$ *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.* ■

### 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.

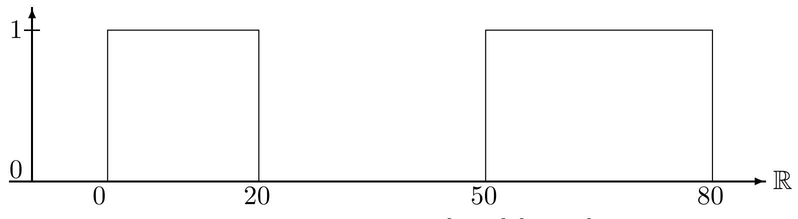
The base set for fuzzy time intervals is the time axis, in FuTIRe represented by the set $\mathbb{R}$ of real numbers. Real numbers allow us to model the continuous time flow which we perceive in our life. A fuzzy time interval in FuTIRe is now a fuzzy subset of the real numbers.

**Definition 2.2.2 (Fuzzy Time Intervals)** *A* fuzzy membership function *in FuTIRe 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*

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

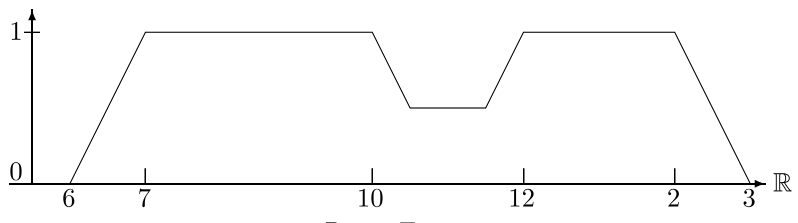
*Given a fuzzy interval $i$ we usually write $i(x)$ to indicate the corresponding membership function. Let $F_\mathbb{R}$ be the set of fuzzy time intervals.* ■

This definition comprises single or multiple crisp intervals like this:



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

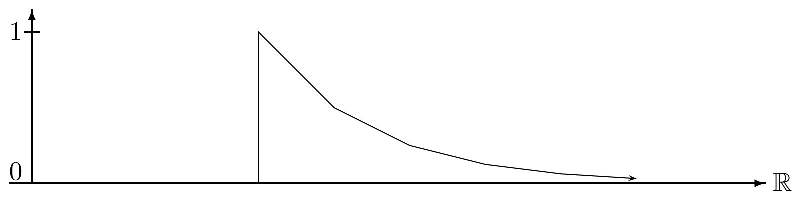
It also comprises finite fuzzy intervals like this one:



*Party Time*

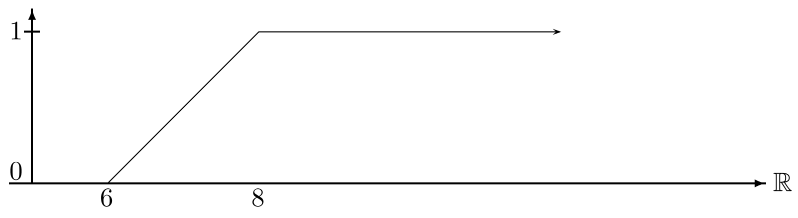
This set may represent a particular party time, where the first guests arrive at 6 pm. At 7 pm all guests are there. Half of them disappear between 10 and 12 pm (because they go to the pub next door to watch an important soccer game). Between 12 pm and 2 am all of them are back. At 2 am the first ones 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.



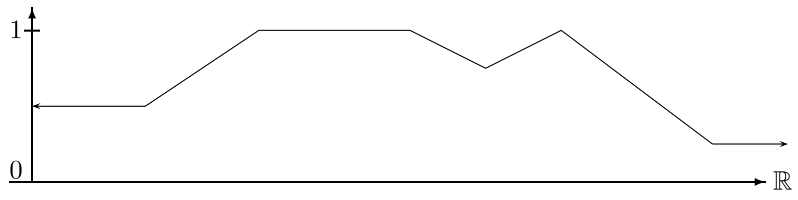
*Dangour Time: Caused by Radioactive Decay*

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 value 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 2.2.3** *The representation of a fuzzy interval as a subset of* $\mathbb{R} \times [0, 1]$ *means that one can apply the standard set operators* $\cap$ *(union),* $\cup$ *(intersection) and* $\setminus$ *(set difference) to fuzzy intervals.*

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

**Definition 2.2.4 (Core, Support and Kernel)**
*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:*

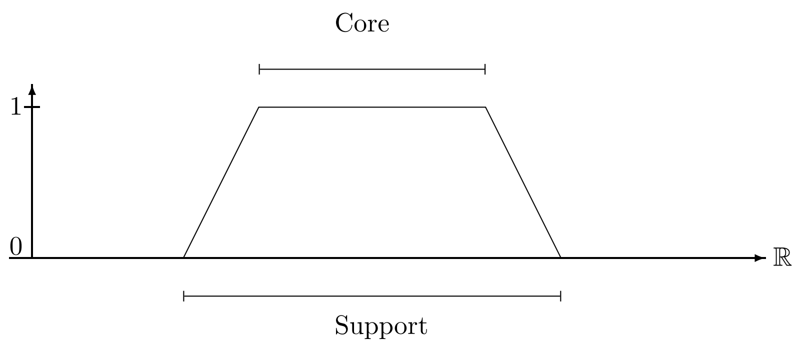
$$S(i) \overset{\text{def}}{=} \{x \in \mathbb{R} \mid i(x) \neq 0\}.$$

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

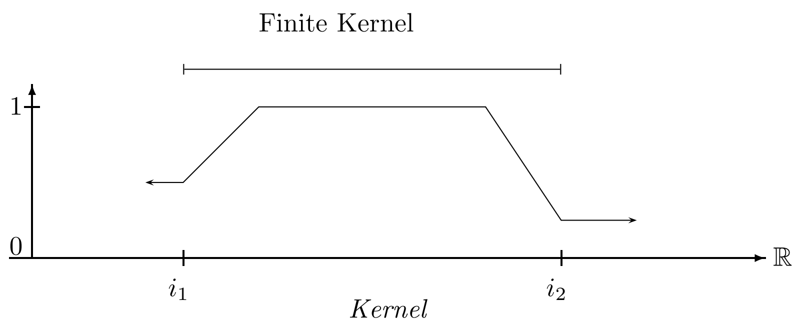
*The* kernel $K(i)$ *of $i$ is the smallest interval $[a, b] \subseteq \mathbb{R}^+$ such that there are $i_1 \in [0, 1]$ and $i_2 \in [0, 1]$ with $i(x) = i_1$ for all $x < a$ and $i(x) = i_2$ for all $x > b$.*

*$K(i)$ can be empty, finite or infinite. If $K(i) = \emptyset$ then $i$ is either empty or infinite and crisp.*

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

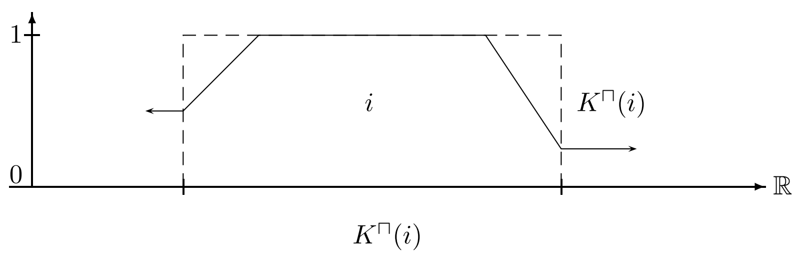


*Core and Support*



*Kernel*

The next picture shows the kernel of the same interval $i$ as crisp set $K^\sqcap(i)$.



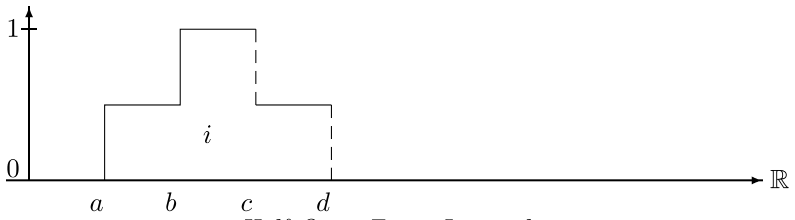
Fuzzy time intervals with finite kernel are of particular interest because although they may be infinite, they can easily be implemented with finite datastructures. Therefore we give them an extra name.

**Definition 2.2.5 (Fuzzy Time Intervals with Finite Kernel)** *Let $F^f_{\mathbb{R}}$ be the set of fuzzy time intervals (Def. 2.2.2) with finite kernel (Def. 2.2.4).* ■

Fuzzy time intervals which are in fact crisp intervals can now be characterized 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. 2.2.2) 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 2.4.24).

### 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*

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

■

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

**Definition 2.2.9 (First and Last Points)** *For an operator $O \in \{C, S, K\}$ and a fuzzy time interval $i$ with $O(i) \neq \emptyset$ let*

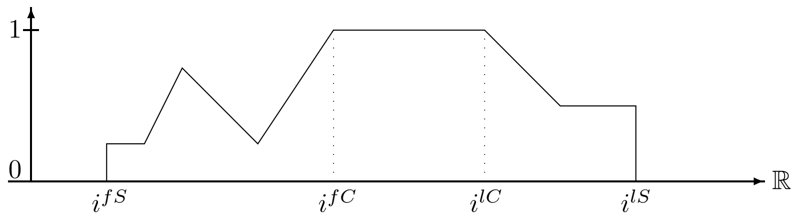
$$i^{fO} \stackrel{\text{def}}{=} \inf(O(i))$$

*be the* first $O$-point *of $i$ and let*

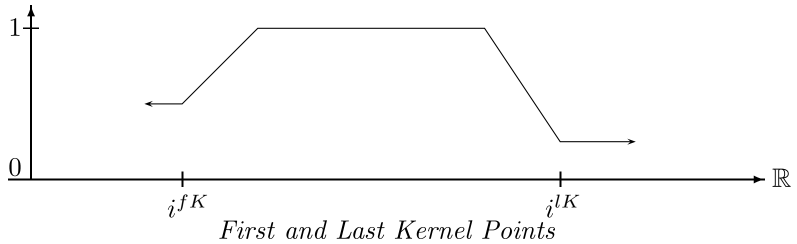
$$i^{lO} \stackrel{\text{def}}{=} \sup(O(i))$$

*be the* last $O$-point *of $i$.* ■

$i^{fC}$ is the first core point, $i^{fS}$ is the first support point, and $i^{fK}$ is the first kernel point. $i^{lC}$ is the last core point, $i^{lS}$ is the last support point, and $i^{lK}$ is the last kernel point.



*First and Last Core and Support Points*



*First and Last Kernel Points*

**Definition 2.2.10 (Height of a Fuzzy Interval)** *For a fuzzy interval $i \in F_{\mathbb{R}}$ let*

$$\hat{i} \stackrel{\text{def}}{=} \sup_x \{i(x)\}$$

*be the* height *of $i$.* ■

$\hat{i}$ is usually, but not necessarily, 1 for nonempty fuzzy time intervals. If $\hat{i} = 0$ then, however, $i$ must be empty.

**Definition 2.2.11 (First and Last Maximal Point)** *If $i \in F_{\mathbb{R}}$ let*

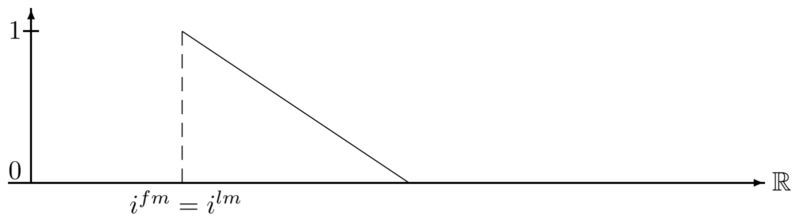
$$i^{fm} \stackrel{\text{def}}{=} \begin{cases} \sup\{x \mid \forall y < x : i(y) < \hat{i}\} & \textit{if the supremum exists} \\ +\infty & \textit{otherwise} \end{cases}$$

*be the* first maximum, *and*

$$i^{lm} \stackrel{\text{def}}{=} \begin{cases} \inf\{x \mid \forall y > x : i(y) < \hat{i}\} & \textit{if the infimum exists} \\ +\infty & \textit{otherwise} \end{cases}$$

*be the* last maximum. ■

The next picture shows an example where $i^{fm} = i^{lm}$ and where $i\hat{}$ is really the supremum, and not the maximum because $i(i^{fm}) = 0$.



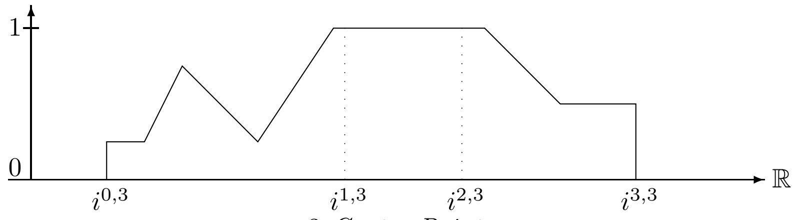
*First and Last Maximal Points*

If $i\hat{} = 1$ then $i^{fm} = i^{fC}$ and $i^{lm} = i^{lC}$ (Def. 2.2.9). If, however, $i\hat{} < 1$ then $i^{fm}$ and $i^{lm}$ have nothing to do with the core of $i$.

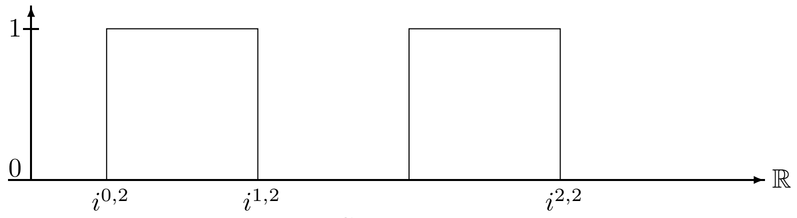
**Center Points**
The $n, m$*-center points* defined below are used to express temporal notions like 'the first half of the year', or 'the second quarter of the year' or more exotic expressions like 'the 25th 49th of the weekend' etc. The notion of $n, m$-center points makes only sense for finite intervals.

**Definition 2.2.12 ($n, m$-Center Points)** *Let $i \in F_{\mathbb{R}}$ with $|i| < \infty$. For two integers $m > 1$ and $0 \leq n \leq m$ we define the $n, m$-*center 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-Center Points*



*$n$, 2-Center Points*

**Middle Points**:
The middle point between the center 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 $i^{3,4}$.

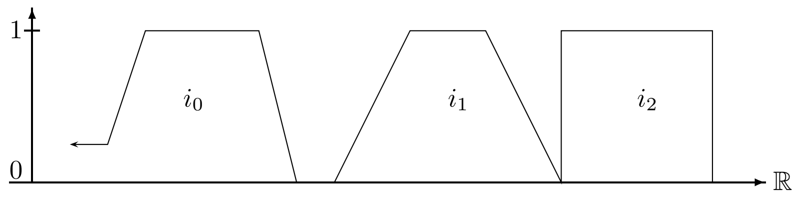
**Components**
Fuzzy time intervals can consist of several different components. A component is a sub-interval of a fuzzy interval such that the left and right end is either infinity, or the membership function drops down to 0. Let $Cmp(i)$ be the list of components of $i$. $nComponents(i)$ is the number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

**Definition 2.2.13 (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 2.2.14 (Y-Functions)**
*Y-FCT*$^n \stackrel{\text{def}}{=} \{f : [0,1]^n \mapsto [0,1]\}$ *is the set of n-place* y-functions.
*They map fuzzy values to fuzzy values.*

*Y-FCT*$\stackrel{\text{def}}{=} \bigcup_{n \geq 0}$ *Y-FCT*$^n$. ■

**Definition 2.2.15 (Interval Operators)**
*I-OPs*$^n \stackrel{\text{def}}{=} \{g : F_{\mathbb{R}}{}^n \mapsto F_{\mathbb{R}}\}$ *is the set of n-place* interval operators.
*They map fuzzy intervals to fuzzy intervals.*

*I-OPs*$\stackrel{\text{def}}{=} \bigcup_{n \geq 0}$ *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 2.2.16 (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 of the polygons need to be transformed.

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

**Definition 2.2.17 (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 $(x_1,z_{11})$ – $(x_2,z_{12})$, …, $(x_1,z_{n1})$ – $(x_2,z_{n2})$ 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 characterized in the following way:

**Proposition 2.2.18 (Characterization 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 $n(y) = 1 - y$.

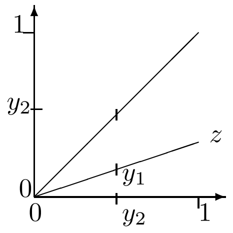
The characterization of two-place linear y-functions is a bit trickier.

**Proposition 2.2.19 (Characterization of Two-Place Linear y-Functions)** *A two-place y-function $f$ is linear if and only if the following condition holds:*

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

**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 $z = y_1/y_2$.

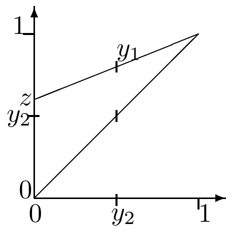


Since $f$ is linear we have

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

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 function. 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 2.2.20 (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 2.2.21 (Standard Negation and $\lambda$-Complement)** *The function*

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

*is the standard fuzzy negation.*

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

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

*Both functions $n$ and $n_\lambda$ are negation functions in the sense of Def. 2.2.20.*

*$N(i)(x) \stackrel{\text{def}}{=} n(i(x))$ is the* standard complement *operator.*
*$N_\lambda(i)(x) \stackrel{\text{def}}{=} n_\lambda(i(x))$ is the* $\lambda$-complement *operator.*

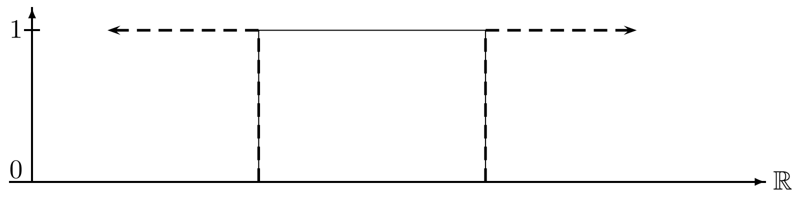
*If $i$ is a crisp interval then $N(i) = N_\lambda(i)$* ■

**Proposition 2.2.22 (Idempotency of the negation functions)** *For every $y \in [0,1]$ we have for the standard negation $n(n(y)) = y$ and for the $\lambda$-complement: $n_\lambda(n_\lambda(y)) = y$.*

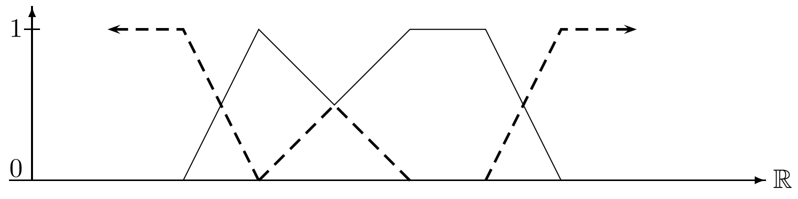
*The proof is straightforward.* ■

This property need not hold for other negation functions.

We give some examples for standard and $\lambda$-complement. The dashed lines indicate the complement.

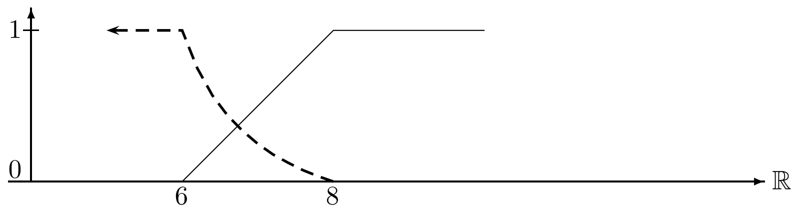


*Standard Complement and $\lambda$-Complement for a Crisp Interval*



*Standard Complement for a Fuzzy Interval*

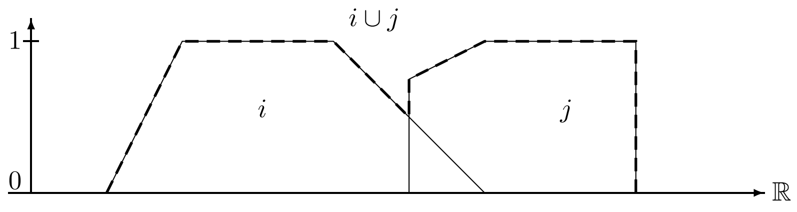
If we define 'tonight' as a fuzzy interval, rising from 0 at 6pm to 1 at 8pm, we could use the standard complement for 'before tonight'. The term 'long before tonight' must of course be represented differently to 'before tonight'. A $\lambda$-complement version with $\lambda = 2$ looks as follows:



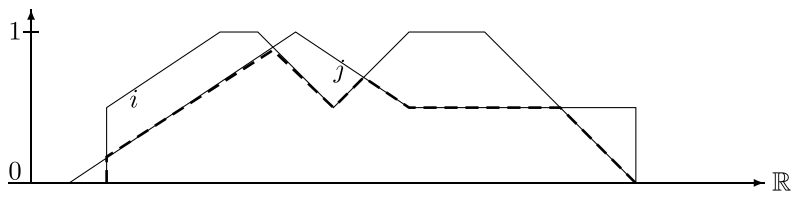
*$\lambda$-Complement for $\lambda = 2$*

If $\lambda$ is increased then the descend from 6 pm till 8 pm becomes steeper. A suitable $\lambda$ 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 conorms, must obey certain axioms in order to satisfy our intuition about union and intersection.

**Definition 2.2.23 (Triangular Norms and Conorms)** *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 conorm *or* t-conorm *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 conorms are y-functions in Y-FCT*$^2$ *(Def. 2.2.14).*

*Let* $TNorm$ *be the set of triangular norms and*
*let* $TCoNorm$ *be the set of triangular conorms.* ■

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

**Definition 2.2.24 (Intersection and Union)** *Let* $i,j \in F_{\mathbb{R}}$ *be two fuzzy intervals. If* $T$ *is a triangular norm and* $S$ *a triangular conorm (Def. 2.2.23) then*

$$(i \cap_T j)(x) \overset{\text{def}}{=} T(i(x),j(x))$$
$$(i \cup_S j)(x) \overset{\text{def}}{=} S(i(x),j(x))$$

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

**Example 2.2.25 (Standard Fuzzy Intersection and Union)** *The function* min *is a triangular norm and the function* max *is a triangular conorm. Therefore*

$$(i \cap_{\min} j)(x) \overset{\text{def}}{=} \min(i(x),j(x))$$
$$(i \cup_{\max} j)(x) \overset{\text{def}}{=} \max(i(x),j(x))$$

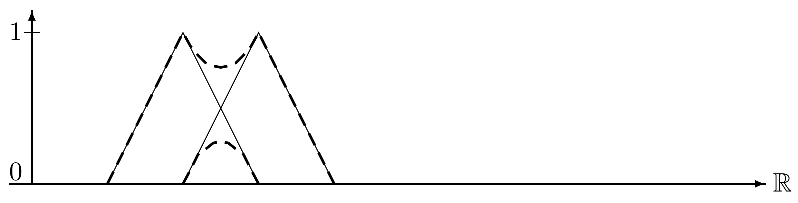
*are the standard fuzzy intersection and union operators.* ■

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

**Example 2.2.26 (Hamacher Family)** *The Hamacher family consists of the following parameterized families of triangular norms and conorms, and negation functions ($\lambda$-complement):*

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

■



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

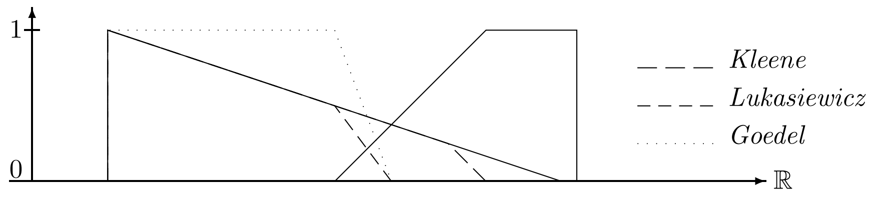
**Set Difference of Fuzzy Time Intervals** Set difference $i \setminus j$ can also be defined by means of y-functions. The following versions are derived from corresponding implication functions:

**Definition 2.2.27 (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 2.2.28 (Set Difference)** *The following picture shows the difference between the three versions of set difference.*



*Set Difference*

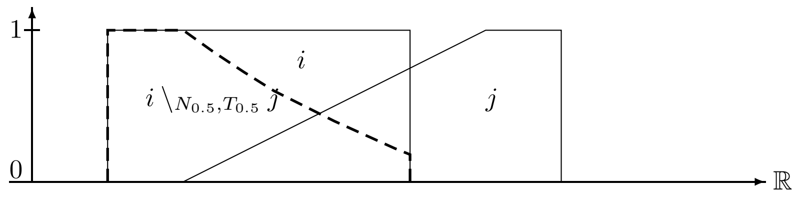
■

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 generalized by replacing $\cap$ with $\cap_T$ and $j^c$ with a complement operator.

**Definition 2.2.29 (Generalized 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*

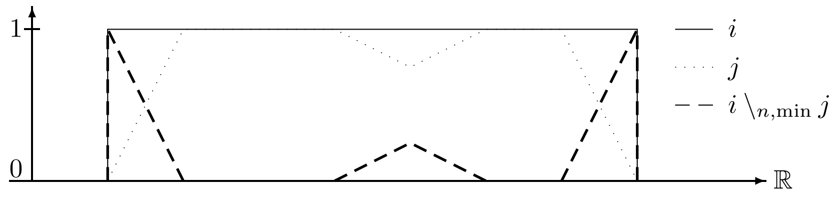
$$(i \setminus_{N,T} j) \overset{\text{def}}{=} i \cap_T N(j)$$

■



*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.

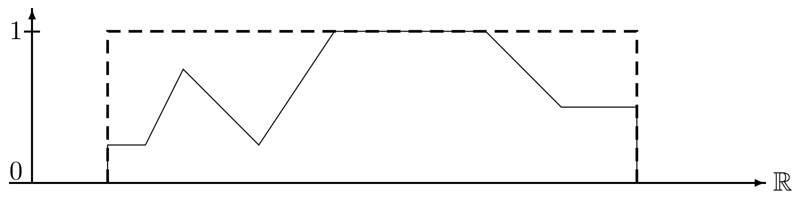


*Set Difference Splits into Three Polygons*

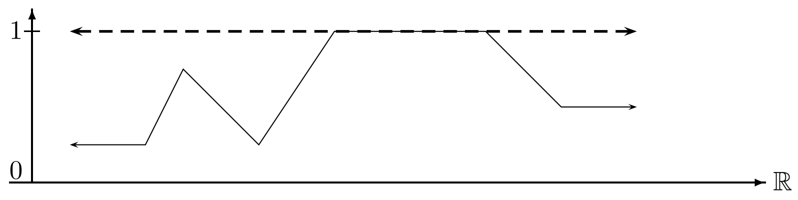
### 2.2.5 Hull Operators for Fuzzy Intervals

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 2.2.30 (Crisp Hull)** *For an interval $i \in F_{\mathbb{R}}$ let $CrH(i)$ be the smallest crisp interval containing $i$.* ∎

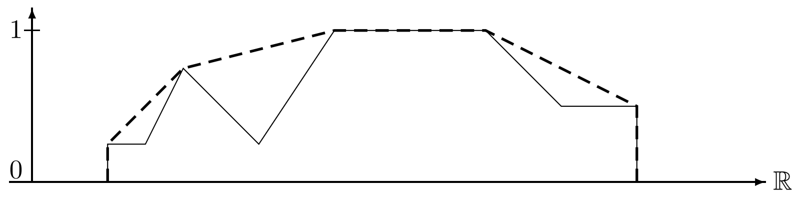


*Crisp Hull of a Finite Interval*

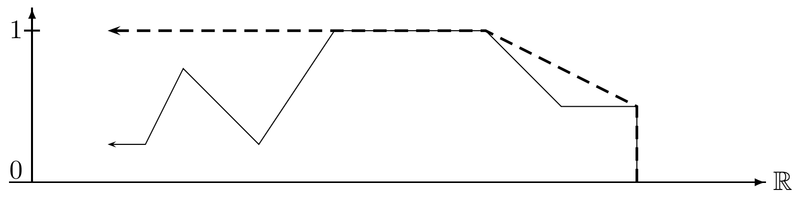


*Crisp Hull of an Infinite Interval*

**Definition 2.2.31 (Convex Hull)** *The* convex hull *$CoH(i)$ of a fuzzy set $i$ is the smallest convex set containing $i$.* ∎



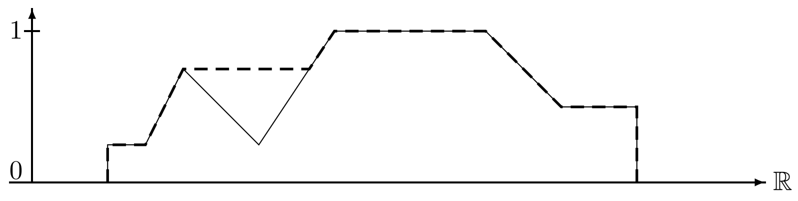
*Convex Hull of a Finite Set*



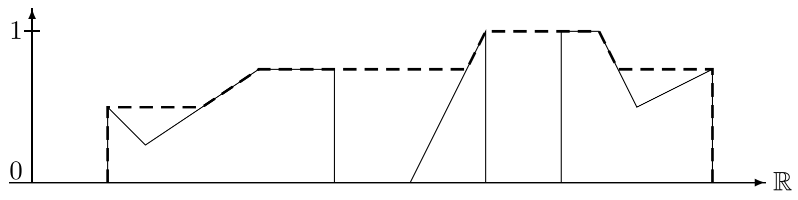
*Convex Hull of an Infinite Set*

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

**Definition 2.2.32 (Monotone Hull)** *The* monotone hull $MoH(i)$ *of a fuzzy set* $i$ *is the smallest* monotone *fuzzy interval containing* $i$*.* Monotone *means that from left to right the fuzzy values* $MoH(i)(x)$ *are rising monotonically to* $\hat{i}$*, and then falling monotonically again.* ■



*Monotone Hull*



*Monotone Hull of a Fuzzy Interval with Three Components*

## 2.2.6 Basic Unary Transformations

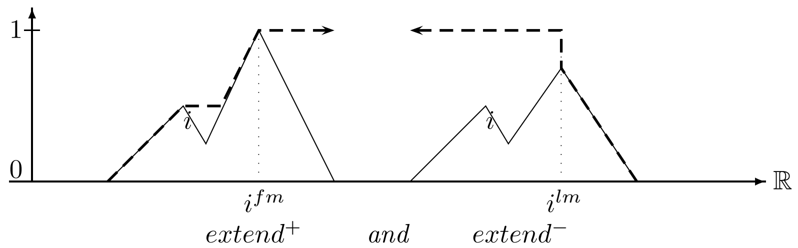
The main purpose of FuTIRe is to provide fuzzy point–interval and interval–interval relations. As we shall see, these relations can also be parameterized, this time with interval operators. Therefore we first introduce a little library of interval operators, which can be used to build the point–interval and interval–interval relations.

**Definition 2.2.33 (Basic Unary Transformations)** *Let $i \in F_{\mathbb{R}}$ be a fuzzy interval. We define the following (parameterized) interval operators:*

$$
\begin{array}{lll}
\mathit{identity}(i) & \stackrel{\text{def}}{=} & i \\[6pt]
\mathit{extend}^{+}(i)(x) & \stackrel{\text{def}}{=} & \begin{cases} \sup_{y \le x} i(y) & \text{if } x \le i^{fm} \\ 1 & \text{otherwise} \end{cases} \\[6pt]
\mathit{extend}^{-}(i)(x) & \stackrel{\text{def}}{=} & \begin{cases} \sup_{y \ge x} i(y) & \text{if } x \ge i^{lm} \\ 1 & \text{otherwise} \end{cases} \\[6pt]
\mathit{scaleup}(i)(x) & \stackrel{\text{def}}{=} & \begin{cases} i(x)/i\hat{} & \text{if } i\hat{} \neq 0 \\ 0 & \text{otherwise} \end{cases} \\[6pt]
\mathit{cut}_{x_1,x_2}(i)(x) & \stackrel{\text{def}}{=} & \begin{cases} 0 & \text{if } x < x_1 \text{ or } x \ge x_2 \\ i(x) & \text{otherwise} \end{cases} \\[6pt]
\mathit{cut}_{x_1,+}(i)(x) & \stackrel{\text{def}}{=} & \begin{cases} 0 & \text{if } x < x_1 \\ i(x) & \text{otherwise} \end{cases} \\[6pt]
\mathit{cut}_{x_1,-}(i)(x) & \stackrel{\text{def}}{=} & \begin{cases} 0 & \text{if } x \ge x_1 \\ i(x) & \text{otherwise} \end{cases} \\[6pt]
\mathit{shift}_n(i)(x) & \stackrel{\text{def}}{=} & i(x-n) \\[6pt]
\mathit{times}_a(i)(x) & \stackrel{\text{def}}{=} & \min(1, a \cdot i(x)) \qquad a \ge 0 \\[6pt]
\mathit{exp}_e(i)(x) & \stackrel{\text{def}}{=} & i(x)^e \qquad e \ge 0 \\[6pt]
\mathit{integrate}^{+}(i)(x) & \stackrel{\text{def}}{=} & \lim\limits_{a \mapsto \infty} \dfrac{\int_{-a}^{x} i(y)dy}{\int_{-a}^{+a} i(y)dy} \\[6pt]
\mathit{integrate}^{-}(i)(x) & \stackrel{\text{def}}{=} & \lim\limits_{a \mapsto \infty} \dfrac{\int_{x}^{+a} i(y)dy}{\int_{-a}^{+a} i(y)dy} \\[6pt]
\mathit{negate}_{\mathit{offset}}(i)(x) & \stackrel{\text{def}}{=} & 1 - i(x - \mathit{offset}) \\[6pt]
\mathit{invert}(i)(x) & \stackrel{\text{def}}{=} & \begin{cases} 1 - i(x) & \text{if } i_k^{fm} \le x < i_{k+1}^{lm} \\ & \text{where } i_0, \dots i_m \text{ are the components of } i \\ 0 & \text{otherwise.} \end{cases}
\end{array}
$$

■

**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*$^{+}(i)$.

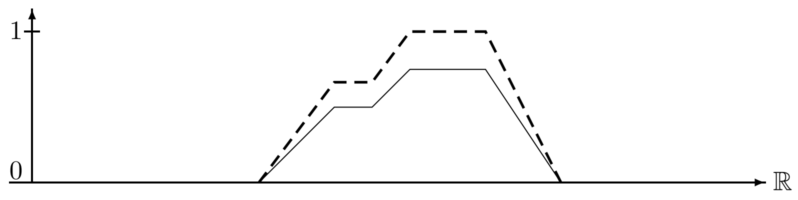


*extend*$^+$ *and* *extend*$^-$

*extend*$^+(i)$ is useful for implementing a '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**

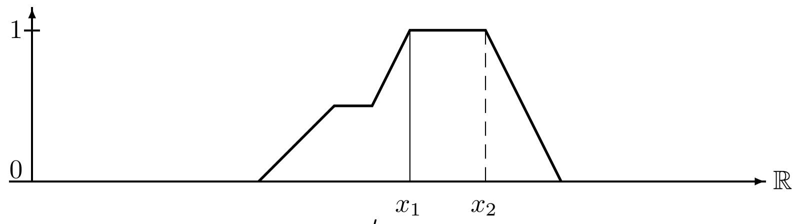
The *scaleup*-function is different to the *identity* function only if the hight $i\hat{}$ is not 1. In this case it scales the membership function up such that *scaleup*$(i)\hat{} = 1$.



*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 $x_2$.

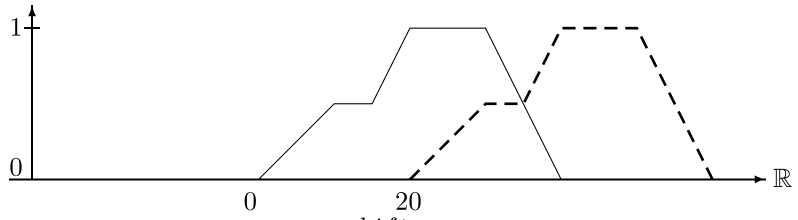


$cut_{x_1,x_2}$

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

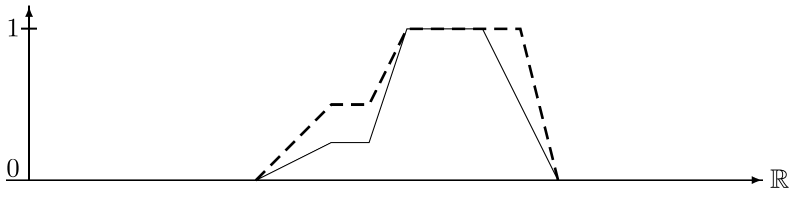
**shift**

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



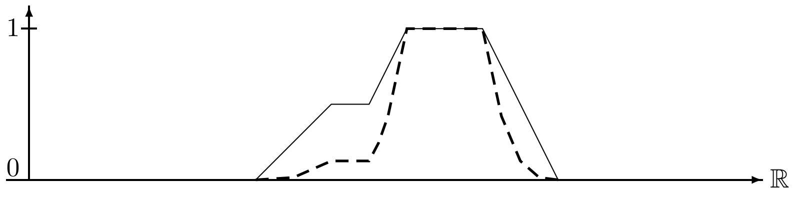
$shift_{20}$

**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.



$times_2$

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



$exp_3$

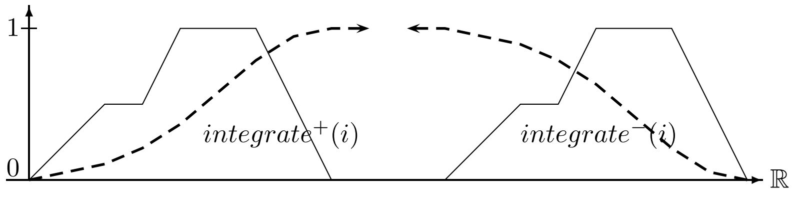
**integrate**
This operator integrates over the membership function and normalizes the integral to values $\leq$ 1. The two integration operators $integrate^+$ and $integrate^-$ can be simplified for finite fuzzy time intervals.

**Proposition 2.2.34 (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 2.2.35 (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*

$$integrate^+(i)(x) = \frac{i_1}{i_1 + i_2} \qquad and \qquad integrate^-(i)(x) = \frac{i_2}{i_1 + i_2}$$

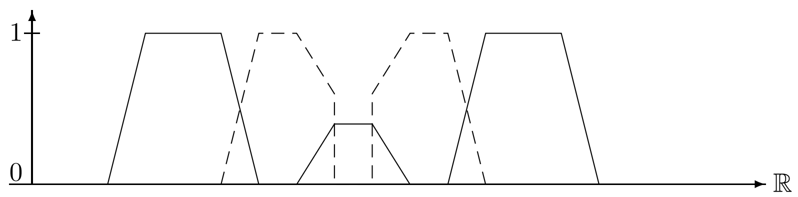
**Proof:**

$$\begin{aligned}
integrate^+(i)(x) &= \lim_{a\mapsto\infty} \frac{\int_{-a}^{x} i(y)dy}{\int_{-a}^{+a} i(y)dy} \\
&= \lim_{a\mapsto\infty} \frac{|i|_{-a}^{i^{fK}} + |i|_{i^{fK}}^{x}}{|i|_{-a}^{i^{fK}} + |i|_{i^{fK}}^{i^{lK}} + |i|_{i^{lK}}^{a}} \\
&= \lim_{a\mapsto\infty} \frac{|i|_{-a}^{i^{fK}}}{|i|_{-a}^{i^{fK}} + |i|_{i^{lK}}^{a}} \\
&= \lim_{a\mapsto\infty} \frac{(i^{fK}+a)\cdot i_1}{(i^{fK}+a)\cdot i_1 + (a-i^{lK})\cdot i_2} \\
&= \lim_{a\mapsto\infty} \frac{a\cdot i_1}{a\cdot i_1 + a\cdot i_2} \\
&= \frac{i_1}{i_1+i_2}
\end{aligned}
\qquad
\begin{aligned}
integrate^-(i)(x) &= \lim_{a\mapsto\infty} \frac{\int_{x}^{+a} i(y)dy}{\int_{-a}^{+a} i(y)dy} \\
&= \lim_{a\mapsto\infty} \frac{|i|_{x}^{i^{lK}} + |i|_{i^{lK}}^{a}}{|i|_{-a}^{i^{fK}} + |i|_{i^{fK}}^{i^{lK}} + |i|_{i^{lK}}^{a}} \\
&= \lim_{a\mapsto\infty} \frac{|i|_{i^{lK}}^{a}}{|i|_{-a}^{i^{fK}} + |i|_{i^{lK}}^{a}} \\
&= \lim_{a\mapsto\infty} \frac{(a-i^{lK})\cdot i_2}{(i^{fK}+a)\cdot i_1 + (a-i^{lK})\cdot i_2} \\
&= \lim_{a\mapsto\infty} \frac{a\cdot i_2}{a\cdot i_1 + a\cdot i_2} \\
&= \frac{i_2}{i_1+i_2}
\end{aligned}$$

■

**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 the *in_the_gap* operator.



*invert*$(i)$

**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 FuTIRe 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 2.2.36 (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:*

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

*If the offset is nonzero we have*

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

*The 'back' linear fuzzification function is:*

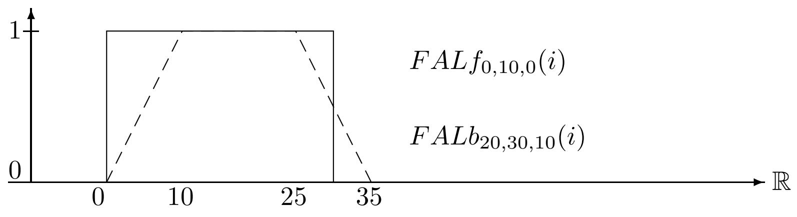
$$FALb_{x_1,x_2,0}(x) \overset{\text{def}}{=} \begin{cases} 0 & \textit{if } x \geq x_2 \\ i(x) & \textit{if } x < x_1 i(x)\frac{x_2-x}{x_2-x_1} \quad \textit{otherwise} \end{cases}$$

*If the offset is nonzero we have*

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

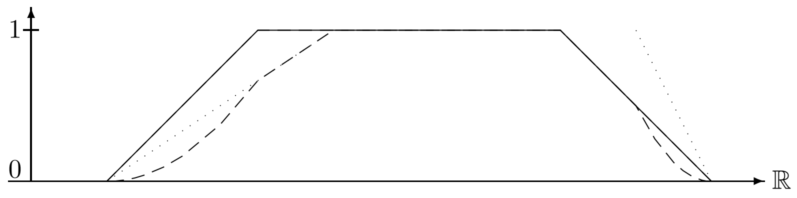
■

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.



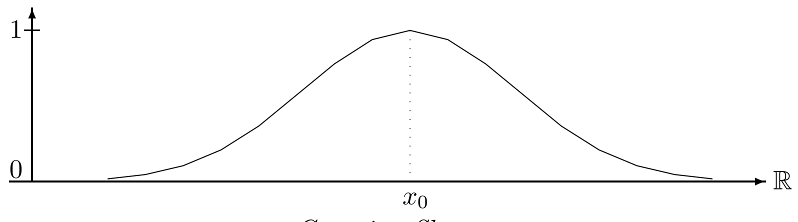
*Linear Fuzzification:*

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*

Besides linear fuzzification, FuTIRe 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 $\sigma$ 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 2.2.37 (Gaussian Fuzzification Function)** *Let $i \in F_{\mathbb{R}}$, $x_h$, $x_0$ and offset be $x$-coordinates.*

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

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

*If the offset is nonzero we have*

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

*The 'back' Gaussian fuzzification function is:*

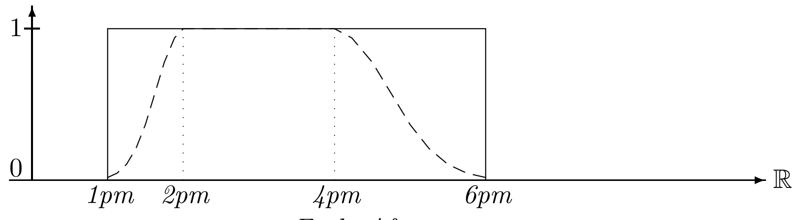
$$FAGb_{x_1,x_2,0}(x) \overset{\text{def}}{=} \begin{cases} 0 & \textit{if } x > 3x_h - 2x_0 \\ i(x) & \textit{if } x < x_0 i(x)e^{-((x-x_0)/\sigma)^2} \quad \textit{otherwise} \end{cases}$$

*If the offset is nonzero we have*

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

■

**Example 2.2.38** *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*

■

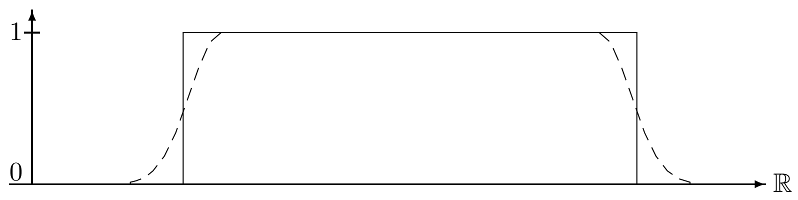
Fuzzification functions with absolute coordinates are not that useful because usually one does not know the coordinates in advance. Therefore FuTIRe also provides fuzzification functions where the parameters are percentage values. $FRLf_{10,5}$, for example, means linear fuzzification where the linear increase is 10% of the kernel size and the offset is 5% of the kernel size (cf. Ex. 2.2.50). $FRLf_{10}$ 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 $i_0$ and the offset is such that $x_h$ coincides with $i^{fK}$.

**Definition 2.2.39 (Fuzzification with Relative Coordinates)** *For an interval $i$, percentage numbers $r$ and $o$ between 0 and 100 we define the relative fuzzification functions.*

*Let $d = (i^{lK} - i^{fK})/100$.*

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

■



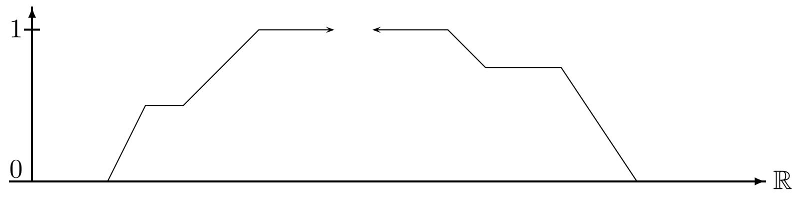
*Relative Gaussian Fuzzification* $FRGf_{20} \circ FRGb_{20}$

**Classification of Interval Operators**

Interval operators which turn fuzzy time intervals into infinite intervals which rise until 1 and then stay constant are of particular interest for the definition of certain point–interval relations (Sect. 2.2.7 below). Therefore we define rising (and falling) fuzzy time intervals and interval operators.

**Definition 2.2.40 (Rising and Falling Fuzzy Intervals and Interval Operators)** *A Fuzzy set $i$ is* rising *iff for its membership function $i(x) = 1$ for all $x > i^{fm}$. $i$ is* falling *iff for its membership function $i(x) = 1$ for all $x < i^{lm}$.*

*An interval operator $f$ is rising iff $f(i_1, \ldots, i_n)$ is* rising *for all tuples $i_1, \ldots, i_n$ of intervals. $f$ is falling iff $f(i_1, \ldots, i_n)$ is* falling *for all tuples $i_1, \ldots, i_n$ of intervals.* ■



*Rising and Falling Intervals*

**Proposition 2.2.41** *The basic unary transformations extend$^+$ and int$^+$ are rising interval operators and the unary transformations extend$^-$ and int$^-$ are falling interval operators.*

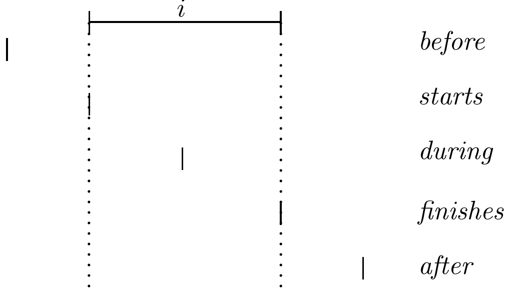
*Any composition $f_1 \circ \ldots \circ f_n \circ f$ where $f$ is a rising (falling) interval operator is again a rising (falling) interval operator.*

*The proofs are straightforward.* ■

### 2.2.7 Point–Interval Relations

There are five basic relations between a time point and a *crisp* interval $i$:

**Definition 2.2.42 (Crisp Point–Interval Relations)**



■

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'. For lists of intervals there are even more point–interval relations, for example 'between the intervals' or 'between the second and third interval' etc.

If we want to 'fuzzify' such relations we face two problems:
1. How should a relation like 'before' work for a fuzzy time interval? For example, what does 'before early afternoon' mean, where 'early afternoon' is represented by a fuzzy set?
2. How can we fuzzify a point–interval relation even for crisp intervals. Can we, for example, obtain a non-zero fuzzy value for the expression 'the party starts *after* midnight', even if the party starts, say, one minute before midnight?

There are no general solutions to these problems. Therefore FuTIRe provides a framework where application specific versions of such relations can be defined and used. Point–interval relations are in this framework represented by (parameterized) interval operators. This may be a bit surprising because relations are in general not functions. The relations are therefore represented indirectly as functions. Consider a point $p$ and an interval $i$. As the result of the relation $p$ *before* $i$ we want a fuzzy value. To get this fuzzy value, we can turn $i$ into another interval *before*$(i)$ such that *before*$(i)(x)$ is the desired fuzzy value.

Since *before*$(\ldots)$ can operate on any fuzzy time interval we have also solved the problem to get a *before* relation between points and *fuzzy* time intervals.

Unfortunately there is no unique definition for *before*$(i)$. One way is to parameterize *before* and all the other relations such that application specific relations can be defined very easily. The parameters are again interval operators.

The general definitions are presented first, and then explained in detail.

**Definition 2.2.43 (Point–Interval Relations as Unary Transformations)**
*Let $i \in F_{\mathbb{R}}$ be a fuzzy interval, and let $T$ be a triangular norm (Def. 2.2.23). $n, m, k$ are non-negative integers.*

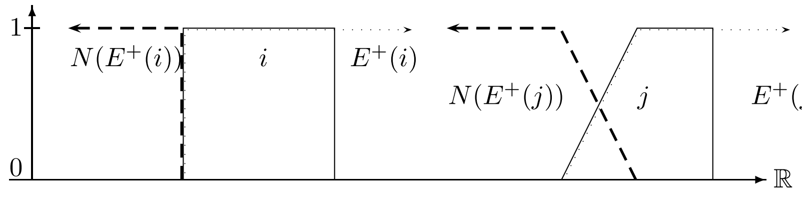
*We define the following point–interval relation operators:*

| | | |
|---|---|---|
| $\text{before}_{N,E^+}(i)$ | $\overset{\text{def}}{=}$ | $\begin{cases} \emptyset & \textit{if } i = \emptyset \\ N(E^+(i)) & \textit{otherwise} \end{cases}$ *where $N$ is a complement operator and $E^+$ a rising operator (Def. 2.2.40).* |
| $\text{after}_{N,E^-}(i)$ | $\overset{\text{def}}{=}$ | $\begin{cases} \emptyset & \textit{if } i = \emptyset \\ N(E^-(i)) & \textit{otherwise} \end{cases}$ *where $N$ is a complement operator and $E^-$ a falling operator (Def. 2.2.40).* |
| $\text{starts}_{E^+,B,T}(i)$ | $\overset{\text{def}}{=}$ | $\textit{scaleup}(E^+(i) \cap_T B(i))$ *where $E^+$ is a rising operator and $B$ a* before *operator* |
| $\text{finishes}_{E^-,A,T}(i)$ | $\overset{\text{def}}{=}$ | $\textit{scaleup}(E^-(i) \cap_T A(i))$ *where $E^-$ is a falling operator and $A$ an* after *operator* |
| $\text{during}_{U,O}(i)$ | $\overset{\text{def}}{=}$ | $U_{l \in Cmp(i)} O(l)$ *where $U$ is a union operator and $O$ any unary transformation* |
| $\text{during}_{D,n,m}(i)$ | $\overset{\text{def}}{=}$ | $D(cut_{i^{n,m},i^{n+1,m}}(i))$ *where $D$ is a* during *operator* |
| $\text{in\_the\_middle}_{D,k,n,m}(i)$ | $\overset{\text{def}}{=}$ | $D(cut_{i^{2^k(2n+1)-1,2^k 2m},i^{2^k(2n+1)+1,2^k 2m}}(i))$ *where $D$ is a* during *operator* |
| $\text{in\_the\_gap}_D(i)$ | $\overset{\text{def}}{=}$ | $D(invert(i))$ *where $D$ is a* during *operator* |
| $\text{in\_the\_}k^{th}\text{gap}_{D,k}(i)$ | $\overset{\text{def}}{=}$ | $D(Component(invert(i),k))$ *where $D$ is a* during *operator* |

*For a particular point $x$ one can get the fuzzy value of the relation $xri$ by applying the corresponding relation operator $O_r$ to $i$ and then calculating $O_r(i)(x)$.* ■

**Before** and **After**
The definition of *before* is $\textit{before}_{n,E^+}(i) \overset{\text{def}}{=} N(E^+(i))$, where $E^+$ is a rising interval operator (Def. 2.2.40) and $N$ is a negation interval operator. $E^+$ projects the rising part of $i$ out and hides all the rest of $i$. $N$ complements the rising part.

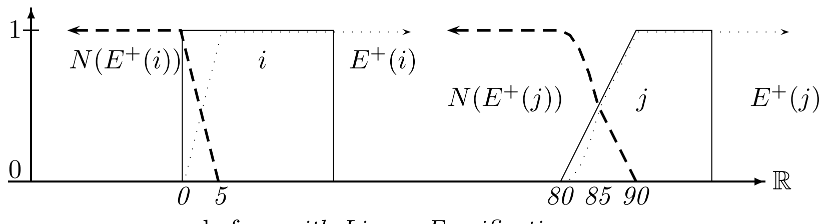
**Example 2.2.44 (*before* with Standard Negation and $E^+ = \textit{extend}^+$)** *The first example gives us the standard* before*-relation. For crisp sets it is the usual before relation. For fuzzy sets it is essentially complements the front part of the interval.*



before *with Standard Negation and extend*$^+$

■

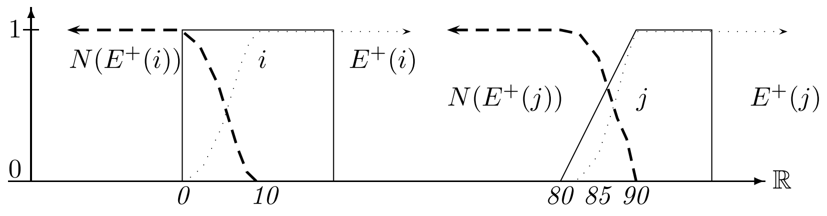
**Example 2.2.45 (More Fuzzy *before*)** *In this example we want to make the* before*-relation a bit more fuzzy, such that points after the start of the interval i get a non-zero fuzzy value. To this end we distort the interval i with linear fuzzification functions $F = FALf_{0,5,0}$ (left graph) and $F = FALf_{80,85,0}$ (right graph). $E^+ \overset{\text{def}}{=} F \circ \textit{extend}^+$ and $N$ is standard complement.*



before *with Linear Fuzzification*

*The fuzzification function has not much of an effect on the second interval because the linear increase ends already in the middle of the ascend from 0 at 80 to 1 at 90.* ■

**Example 2.2.46 (Gaussian *before*)** *This example is similar to the previous one, but instead of a linear fuzzification function we use Gaussian fuzzification functions $F = FAGf_{5,10,0}$ and $F = FAGf_{85,90,0}$ to fuzzify the before relation. $E^+ = F \circ extend^+$ and $N$ is the standard complement.*



before *with Gaussian Fuzzification Function*

■

The *after*-relation is just the symmetric variant of the *before*-relation. Therefore no further explanation is necessary.

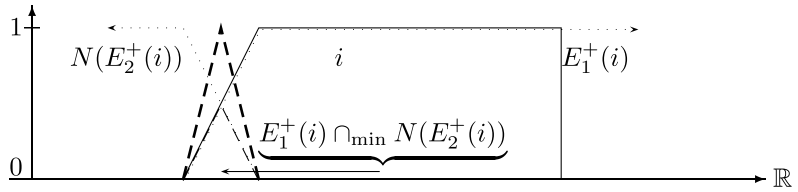
**Starts and Finishes**:
The *finishes* relation is the symmetric variant of the *starts*-relation. Therefore it is sufficient to discuss the *starts*-relation.

The standard *starts*-relation $x$ *starts* $i$ for crisp intervals $i$ is 1 if $x$ 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 basic idea of a fuzzy *starts*-relation is to widen this single peak a little bit. For crisp intervals there are some minor technical problems here. Therefore we first explain it with a fuzzy interval.

The definition is $starts_{E_1^+,E_2^+,T}(i) \overset{\text{def}}{=} scaleup(E_1^+(i) \cap_T N(E_2^+(i)))$.

The rising function $E_1^+$ extracts the front part of $i$. $N(E_2^+(i))$ extracts the same or another front part of $i$ and complements it with the complement function $N$. The first extracted front part and the complemented extracted front part are then intersected. Since the intersection may yield a fuzzy set with maximum fuzzy value $< 1$ it is scaled up to 1.

**Example 2.2.47 (Simple *starts*-Relation)** *In the first example we take $E_1^+ = E_2^+ = extend^+$, the standard complement for $N$, and* min *for the t-norm $T$.*

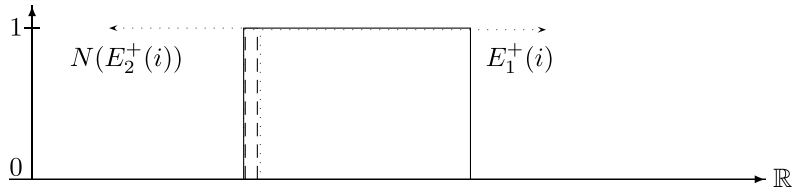


starts

*The dashed line indicates the scaled up intersection.* ■

It is now easy to imagine that the steeper the ascend of the front part of $I$ is, the narrower the peak of the *starts*-relation will be. Unfortunately, in the extreme case where $i$ is a crisp interval, there is no singular *starts*-peak anymore, but the result is just the empty set. If $E_1^+ = E_2^+ = extend^+$ then $E_1^+(i)$ and $N(E_2^+(i))$ are complementary sets, and their intersection is empty. To overcome this problem, FuTIRe just shifts $E_1^+(i)$ by a small amount.

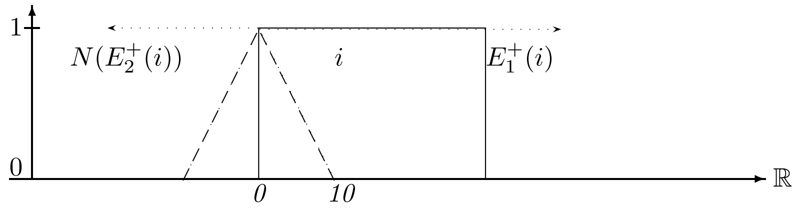
**Example 2.2.48 (*starts*-Relation for a Crisp Interval)** *The dashed line indicates the* starts*-peak.*



starts *for a Crisp Interval* ■

We can now fuzzify the *starts*-relation for crisp intervals by widening the *starts*-peak again.

**Example 2.2.49 (*starts*-relation for a Crisp Interval)** *We choose* $E_1^+ \overset{\text{def}}{=} extend^+ \circ FALf_{0,10,-10}$ *and* $E_2^+ \overset{\text{def}}{=} extend^+ \circ FALf_{0,10,0}$.

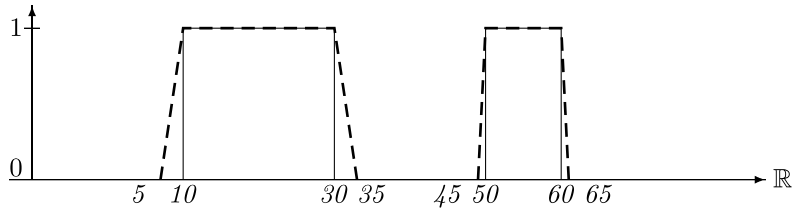


*fuzzy* starts *for a Crisp Interval* ■

**During**:
The simplest version of the *during* operator is just the identity: $x$ *during* $i = identity(i)(x) = i(x)$. This returns 0 for all $x$ outside $i$ and the fuzzy membership value for all $x$ inside $i$. In particular for crisp intervals this is a very strict interpretation of 'during'. We can weaken this strict interpretation by fuzzifying the interval $i$ and taking the membership function of the fuzzified interval. If $i$ consists of one single component, we get then $during(i) \overset{\text{def}}{=} O(i)$ where $O$

fuzzifies $i$ in some way. If $i$ consists of several components, we must fuzzify each component separately and then take the union of all fuzzified components. The final definition is then $during_O(i) \overset{\text{def}}{=} \bigcup_{l \in Cmp(i)} O(l)$.

**Example 2.2.50 (*during* for Crisp Intervals)** *We choose $O = FRLf_{10,10} \circ FRLb_{10,10}$*

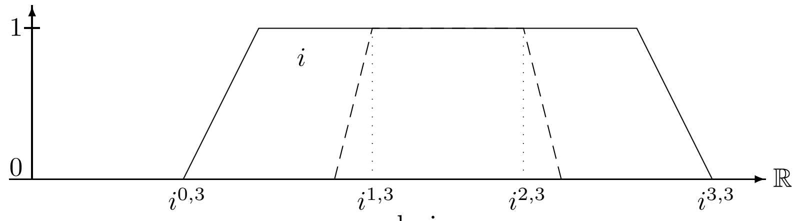


during *for a Crisp Interval with two Components*

■

**During$_{n,m}$**:
The relation 'during the first half' or 'during the second third' or in general 'during the $n^{th}$ $m^{th}$' is very similar to the basic *during*-relation. The only difference is that the corresponding part – the first half or the second third etc. – has to be cut out of the fuzzy interval $i$ before the basic *during* operator is applied. The definition is therefore just $during_{O,n,m}(i) \overset{\text{def}}{=} during_O(cut_{i^{n,m}, i^{n+1,m}})$.

**Example 2.2.51 (*during*$_{1,3}$)** *We choose again $O = FRLf_{10,10} \circ FRLb_{10,10}$.*



during$_{1,3}$

*The dashed line indicates* during$_{1,3}(i)$.

■

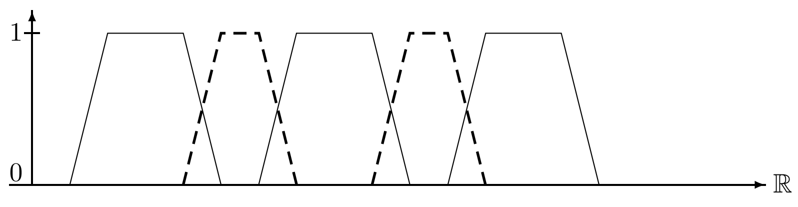
**In the middle**:
This relation focuses on the middle point of the interval between $i^{n,m}$ and $i^{n+1,m}$. The middle point is $i^{2n+1,2m}$. The idea is to cut a slice around this middle point out of $i$ and apply a *during* operator to this slice. The size of the slice is determined by the parameter $k$. The larger $k$ the smaller the slice. This yields the definition

$$in\_the\_middle_{D,k,n,m}(i) \overset{\text{def}}{=} D(cut_{i^{2^k(2n+1)-1,2^k 2m}, i^{2^k(2n+1)+1,2^k 2m}}(i))$$

**In the gap**:
The *in_the_gap* operator is sensitive to the gaps between different components of the fuzzy interval $i$. For example, if $i$ is the time of a soccer game, then *in_the_gap* detects the break between the two halfs of the game. The operator is quite simple: the first step is to invert the interval $i$ in order to make the gaps visible. Then we apply a *during* operator to the inverted $i$.

$$in\_the\_gap_D(i) \overset{\text{def}}{=} D(invert(i))$$

**Example 2.2.52 (*in_the_gap*)** *with $D = identity$*



in_the_gap$_{id}$ ■

The *in_the_gap* operator does not distinguish between the different gaps in the fuzzy interval $i$. Therefore we introduced the operator $\mathit{in\_the\_k}^{th}\mathit{gap}_{D,k}$ with the extra parameter $k$ to select the particular gap. $\mathit{in\_the\_k}^{th}\mathit{gap}_{D,0}$ chooses the first gap, $\mathit{in\_the\_k}^{th}\mathit{gap}_{D,1}$ chooses the second gap etc.

The definition is $\mathit{in\_the\_k}^{th}\mathit{gap}_{D,k}(i) \stackrel{\text{def}}{=} D(\mathit{Component}(\mathit{invert}(i), k))$.

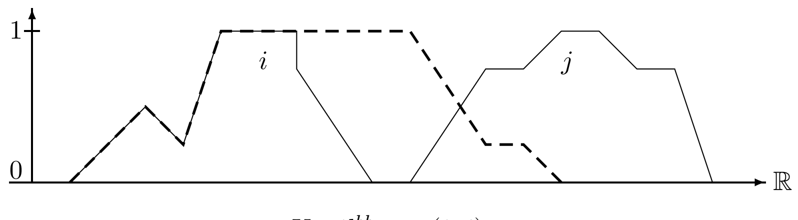
## Until

The 'Until' operator is known from temporal logics [22]. $\varphi$ *Until* $\psi$ in a temporal logic usually means 'eventually $\psi$ holds and $\varphi$ holds *until* this time point. FuTIRe also provides an 'Until' operator, but $\varphi$ and $\psi$ are not formulae, but fuzzy time intervals. With FuTIRe's 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. FuTIRe provides all four combinations.

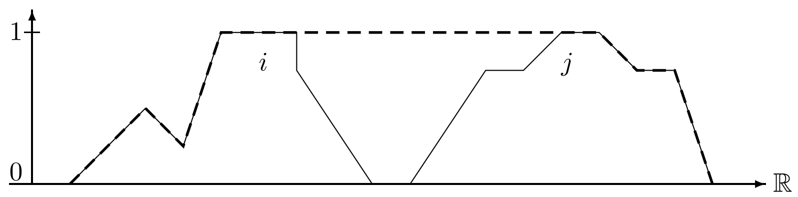
**Definition 2.2.53 (Until)** *Let $E^+$ be a rising function (Def. 2.2.40), $E^-$ a falling function, $C$ a complement operator, and $T$ a t-norm (Def. 2.2.23). For two fuzzy sets $i$ and $j$ we define*

$$\begin{array}{lcl}
\mathit{Until}^{bb}_{E^+,T,C}(i,j) & \stackrel{\text{def}}{=} & E^+(i) \cap_T C(E^+(j)) \\
\mathit{Until}^{be}_{E^+,E^-,T}(i,j) & \stackrel{\text{def}}{=} & E^+(i) \cap_T E^-(j) \\
\mathit{Until}^{eb}_{E^+,E^-,T,C}(i,j) & \stackrel{\text{def}}{=} & C(E^-(i)) \cap_T C(E^+(j)) \\
\mathit{Until}^{ee}_{E^-,T,C}(i,j) & \stackrel{\text{def}}{=} & C(E^-(i)) \cap_T E^-(j).
\end{array}$$

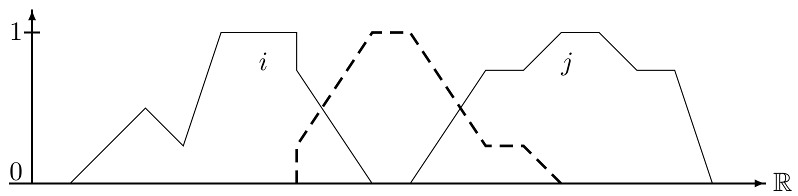
The next four figures show the four cases for the *Until* operator when two single non-overlapping fuzzy intervals are involved. We choose $E^+ = \mathit{extend}^+$, $E^- = \mathit{extend}^-$, $C$ is the standard complement and $T = \min$.



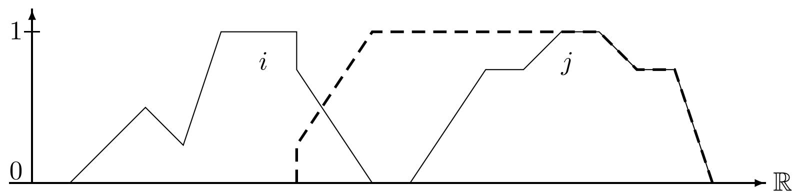
$\mathit{Until}^{bb}_{E^+,T,C}(i,j)$



$Until^{be}_{E^+,E^-,T}(i,j)$

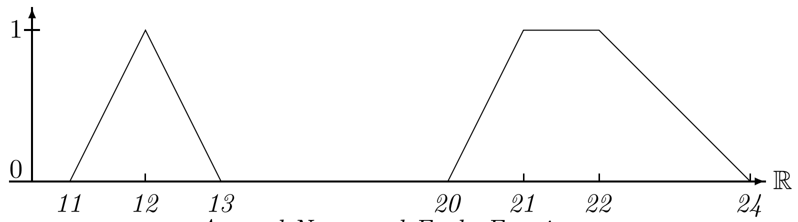


$Until^{eb}_{E^+,E^-,T,C}(i,j)$



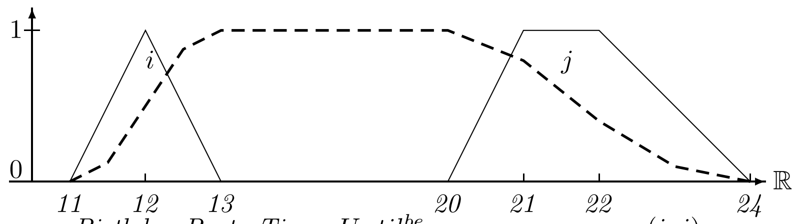
$Until^{ee}_{E^-,T,C}(i,j)$

**Example 2.2.54 (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 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. Therefore we choose $E^+ = integrate^+$ and $E^- = integrate^-$ (Def. 2.2.33). The resulting fuzzy set is:*



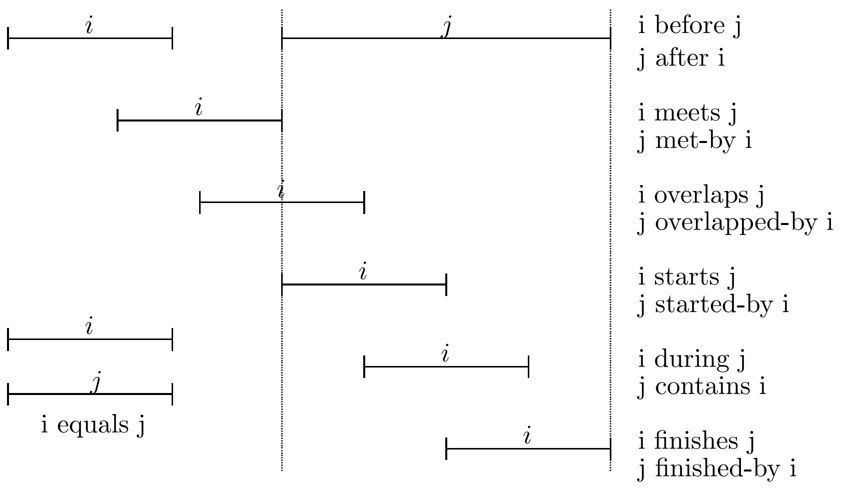
*Birthday Party Time:* $Until^{be}_{integrate^+,integrate^-,N}(i,j)$

*The dashed curve may, for example, represent the percentage of people at the party at a give time.* ∎

### 2.2.8 Interval–Interval Relations

Allen's seven interval relations [5] are the basic relations between two crisp intervals.

**Definition 2.2.55 (Allen's Interval–Interval Relations)**



∎

We want to generalize these relations in two ways:

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 0;
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 generalized 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. Take for example the point–interval relation *before*. We get the fuzzy value for a point $x$ and an interval $j$ by evaluating *before*$(j)(x)$. If we extend the point $x$ to an interval $i$, we can obtain an average value for $i$ *before* $j$ by integrating over $i$ and normalizing the result with $|i|$: $before(i,j) = \int i(x) \cdot before(j)(x)\ dx/|i|$.

The exact definition below is a bit more complicated because it takes into account that $i$ may be empty or infinite.

We summerize the interval–interval relations in the definition below and then explain them in detail one by one.

**Definition 2.2.56 (Interval–Interval Relations)**
*Let $i$ and $j$ be two fuzzy time intervals. We need the following point–interval operators:*
*$B$ is a Point–interval* before *operator (Def. 2.2.43),*
*$E^+$ is a rising transformation (like extend$^+$) (Def. 2.2.40),*
*$D$ is a Interval–interval* during *operation (defined below)*
*$D_p$ is a point–interval* during *operator (Def. 2.2.43),*
*$S, S_1$ and $S_2$ are point–interval* starts *operators (Def. 2.2.43),*
*$F, F_1$ and $F_2$ are point–interval* finishes *operators (Def. 2.2.43).*
*For all 6* starts *and* finishes *operators $O$ it is assumed that $|O(i)| < \infty$ for all fuzzy time intervals $i$.*

*The interval–interval relations are defined as follows:*

$$\text{before}_B(i,j) \stackrel{\text{def}}{=} \begin{cases} 0 & \textit{if } i \textit{ is empty or positive infinite or } j \textit{ is empty} \\ 1 & \textit{if } i \textit{ is negative infinite and } i \cap_{min} j = \emptyset \\ \int (i \cap_{min} j)(x) \cdot B(j)(x)\ dx/|i \cap_{min} j| & \textit{if } i \textit{ is negative infinite} \\ \int i(x) \cdot B(j)(x)\ dx/|i| & \textit{otherwise} \end{cases}$$

$$\text{meets}_{F,S}(i,j) \stackrel{\text{def}}{=} \begin{cases} 0 & \textit{if } i \textit{ or } j \textit{ are empty or } i \textit{ is positive infinite or } j \textit{ is negative infinite} \\ \int F(i)(x) \cdot S(j)(x)\ dx/N(F(i), S(j)) & \textit{otherwise} \end{cases}$$

$$\text{during}_{D_p}(i,j)$$
$$\stackrel{\text{def}}{=} \begin{cases} 1 & \textit{if } i \textit{ is empty} \\ 0 & \textit{if } j \textit{ is empty} \\ (i(-\infty) \cdot j(-\infty) + i(+\infty) \cdot j(+\infty))/(i(-\infty) + i(+\infty)) & \textit{if } i \textit{ is infinite} \int i(x) \cdot D_p(j)(x)\ dx/|i| \quad \textit{otherwise} \end{cases}$$

$$\text{overlaps}_{E+,D}(i,j) \stackrel{\text{def}}{=} \begin{cases} 0 & \textit{if } i \textit{ or } j \textit{ is empty or } j \textit{ is negative infinite} \\ i^{fS} < j^{fS} & \textit{if } i \textit{ is positive infinite (as Boolean result)} \\ i^{lS} < j^{fS} & \textit{if } j \textit{ is positive infinite (as Boolean result)} \\ (1 - D(i, E^+(j))) \cdot D(i,j)/N'(i,j) & \textit{otherwise} \end{cases}$$
*where* $N'(i,j) \stackrel{\text{def}}{=} \max_a((1 - D(shift_a(i), E^+(j))) \cdot D(shift_a(i), j))$

$$\text{starts}_{S_1,S_2,D}(i,j) \stackrel{\text{def}}{=} \begin{cases} 0 & \textit{if } i \textit{ or } j \textit{ are empty or one of } i \textit{ and } j \textit{ is negative infinite} \\ D(i,j) & \textit{if both } i \textit{ and } j \textit{ are negative infinite} \\ \frac{\int S_1(i)(x) \cdot S_2(j)(x)\ dx}{N(S_1(i), S_2(j))} \cdot D(i,j) & \textit{otherwise} \end{cases}$$

$$\text{finishes}_{F_1,F_2,D}(i,j) \stackrel{\text{def}}{=} \begin{cases} 0 & \textit{if } i \textit{ or } j \textit{ are empty or one of } i \textit{ or } j \textit{ is positive infinite} \\ D(i,j) & \textit{if both } i \textit{ and } j \textit{ are positive infinite} \\ \frac{\int F_1(i)(x) \cdot F_2(j)(x)\ dx}{N(F_1(i), F_2(j)))} \cdot D(i,j) & \textit{otherwise} \end{cases}$$

$$\text{equals}_D(i,j) \stackrel{\text{def}}{=} D(i,j) \cdot D(j,i)$$

*where the normalization factor is* $N(i,j) \stackrel{\text{def}}{=} min(|i|, |j|)$ *or* $N(i,j) \stackrel{\text{def}}{=} \max_a \int i(x-a) \cdot j(x)\ dx$

*Notice that the first factor in the integrals above always denotes a finite interval. This guarantees that the integrals are finite, even if the second term denotes an infinite interval.* ■

**The Normalization Factor**

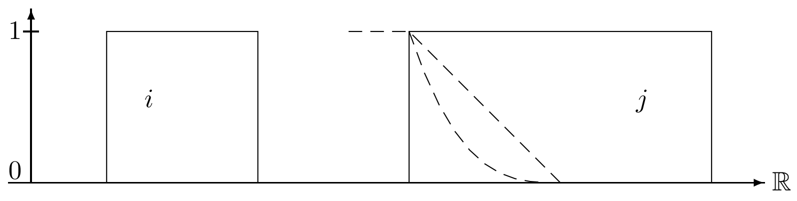
The integrals for the relations *meets*, *starts* and *finishes* are normalization by a factor $N(i, j)$. The optimal choice is $N(i, j) \overset{\text{def}}{=} \max_a \int i(x-a) \cdot j(x)\ dx$ because this normalizes the integral such that the value 1 is a possible result. The intuition behind this is that for *meets*$(i, j)$ there should be a position of $i$ relative to $j$ where *meets*$(i, j) = 1$. For *starts*$(i, j)$ and *finishes*$(i, j)$ there should be a position of $i$ where at least the first factor which determines whether the left/right end of $i$ and $j$ coincide, is 1. $\max_a \int i(x-a) \cdot j(x)\ dx$ as normalization factor guarantees this. Unfortunately, this causes a nontrivial search problem (Sec. 2.2.9). Therefore there is a second choice for the normalization factor: $N(i, j) \overset{\text{def}}{=} min(|i|, |j|)$. This avoids the search problem, but it means that the value of *meets*$(i, j)$ etc. may be always smaller than 1.

We examine the interval–interval relations now in detail.

**Before for Finite Intervals**

The definition for finite intervals is: $before_B(i, j) \overset{\text{def}}{=} \int i(x) \cdot B(j)(x)\ dx/|i|$ 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 normalization factor is just $|i|$. The rationale behind this is the following: if $j$ is finite then $B(j)(x) = 1$ if $x$ is small enough. If we move the interval $i$ into the area where $B(j)(x) = 1$ then the integral becomes $\int i(x) \cdot B(j)(x)\ dx = \int i(x) \cdot 1\ dx = |i|$, such that *before*$(i, j) = 1$. Thus, $|i|$ as normalization factor yields the right result.

**Example 2.2.57 (*before*)** *The next picture illustrates the fuzzy* before *relation for two finite crisp intervals. $B$ is the standard point–interval* before *operator of Example 2.2.44. For this particular operator $B$ and crisp intervals $i$ and $j$* $\text{before}_B(i, j)$ *yields the percentage of points in $i$ which are before $j$ (in the usual sense). The upper dashed line indicates the fuzzy value at position $x$ 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* $\text{before}_B(i, j)$ *is, for example, exponentiated with an exponent $> 1$. The lower dashed line in the figure therefore indicates* $\text{before}_B(i, j)^3$.

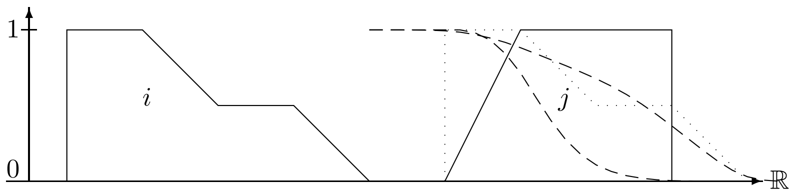


$\text{before}_B(i, j)$ *and* $\text{before}_B(i, j)^3$

■

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 exponent.

**Example 2.2.58 (*before* for Fuzzy Intervals)** *The next picture shows the* before*-relation for real fuzzy intervals. The upper dashed line indicates the result of the* before*-relation at position $x$ if the positive end of the interval $i$ is moved to $x$. 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.*



before *for Fuzzy Intervals*

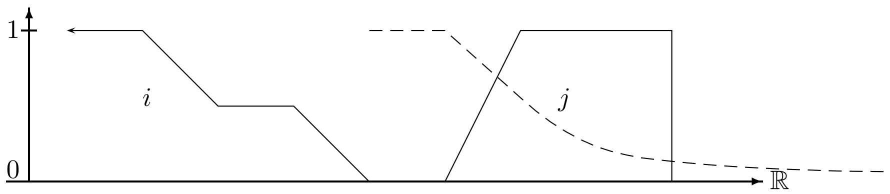
■

**Before for Infinite Intervals**
If the interval $i$ is positive infinite, then nothing can be after $i$. Therefore the relation $\mathit{before}(i,j)$ must yield 0. If the interval $j$ is negative infinite, then nothing can be before $j$. Therefore the relation $\mathit{before}(i,j)$ must also yield 0.

If $i$ is negative infinite and $j$ is finite or positive infinite then $\mathit{before}(i,j)$ may well be not false. The problem is how to measure the degree of 'beforeness' in this case. Since $i$ is infinite, $\int i \cdot B(j)\ dx$ 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 formula is then

$$\mathit{before}_B(i,j) \overset{\text{def}}{=} \int_{min} (i \cap j)(x) \cdot B(j)\ dx / |i \cap_{min} j|.$$

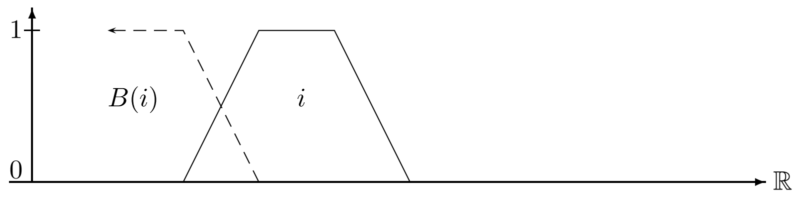
**Example 2.2.59 (*before* for Infinite Intervals)** *This picture shows the development of* before$_B(i,j)$ *when $i$ is negative infinite and its positive end is moved along the x-axis. The fuzzy value drops down, but not to 0. It remains constant after a while.*



before *for Infinite Intervals*

■

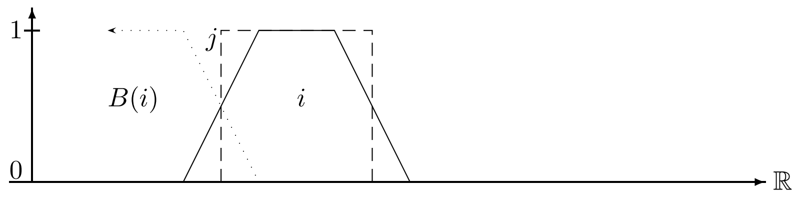
**Properties of the *before* relation for Finite Intervals**: The crisp relation on crisp sets is irreflexive: $\neg\mathit{before}(i,i)$ holds for all $i$. $\mathit{before}_{st}(i,i) = 0$ holds also for crisp sets. $\mathit{before}_{st}(i,i) = 0$ means in this case that 0% of $i$ is before $i$ itself. 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 before $i$ to a certain degree. Therefore $\mathit{before}_{st}(i,i) > 0$.

The crisp *before* relation is asymmetric: $\forall i,j\ \mathit{before}(i,j) \Rightarrow \neg\mathit{before}(j,i)$. A similar property also holds for the $\mathit{before}_{st}$ relation on crisp sets: if a non-zero fraction of $i$ is before $j$ then nothing of $j$ can be before $i$. $\mathit{before}_{st}(i,j) > 0 \Rightarrow \mathit{before}_{st}(j,i) = 0$.



*Counterexample for Asymmetry*

$B(i)$ has a non-zero intersection with $j$ and $B(j)$ has a non-zero intersection with $i$. Therefore $\mathit{before}_{st}(i,j) > 0$ and $\mathit{before}_{st}(j,i) > 0$.

The fuzzy version of transitivity relates $\mathit{before}(i,j)$ and $\mathit{before}(j,k)$ with $\mathit{before}(i,k)$ in some way. There is such a relation for the standard *before* relation and crisp sets: if $\mathit{before}_{st}(i,j) = a$ ($a \cdot 100\%$ of $i$ is before $j$) and $\mathit{before}_{st}(j,k) = b$ ($b \cdot 100\%$ of $j$ is before $k$) then one can show that $\mathit{before}_{st}(i,k) = \min(1, a + b \cdot |i|/|j|)$. Nothing of this kind holds for fuzzy sets.

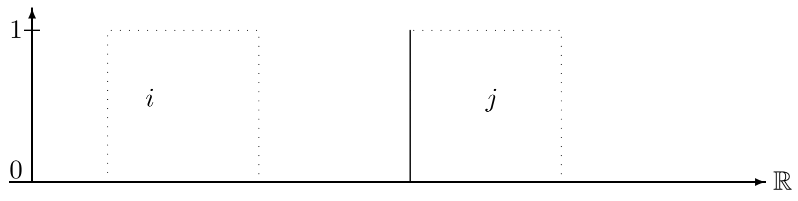
**Meets**

The classical 'meets' relation yields 'true' if the end of the first interval $i$ touches the beginning of the second interval $j$. The back end of $i$ and the front end of $j$ are therefore relevant for evaluating $\mathit{meets}(i,j)$. We can get the back end of $i$ in our fuzzy setting with the point–interval *finishes* operator, and the front end of $j$ with the point–interval *starts* operator (cf. Example 2.2.47). 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 normalizes 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 definition is therefore $\mathit{meets}_{F,S}(i,j) \overset{\text{def}}{=} \int_{-\infty}^{+\infty} F(i)(x) \cdot S(j)(x)\ dx / N(F(i), S(j))$.

The normalization factor $N(F(i), S(j)) \overset{\text{def}}{=} \max_a \int F(i)(x-a) \cdot S(j)(x)\ dx$ amounts to a search problem where $F(i)$ is moved along the $x$-axis to find the position for $i$ where the integral becomes maximal. This guarantees that there is a position for $i$ where $\mathit{meets}(i,j) = 1$.

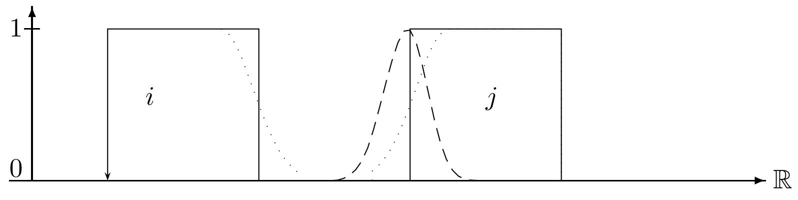
**Example 2.2.60 (*meets* for Crisp Intervals)**

*The first picture shows the* meets*-relation where for crisp intervals the operators $F$ and $S$ have a singular peak. Consequently the* meets*-relation has also a singular peak when the interval $i$ meets $j$ in the crisp sense.*



*Crisp* meets *for Crisp Intervals*

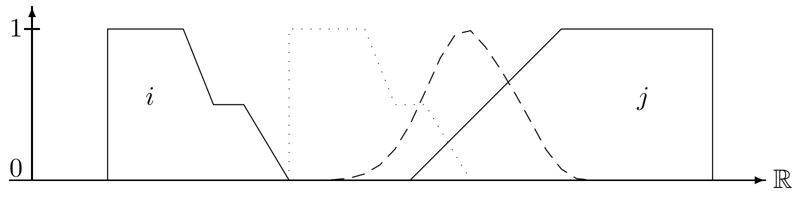
*The next picture shows the result of the* meets-*relation when the* finishes- *and* starts *operators fuzzify the crisp sets. The dotted lines show the fuzzified crisp sets (with Gaussian fuzzification). The dashed line is again the result of* meets *when the endpoint of $i$ is moved along the $x$-axis.*



*Fuzzy* meets *for Crisp Intervals*

■

**Example 2.2.61 (*meets* for Fuzzy Intervals)** *We illustrate the fuzzy* meets-*relation with two fuzzy time intervals and the simple point–interval* finishes *and* starts *operators of Example 2.2.47. The dashed line shows the results of the* meets-*relation when the interval $i$ is moved along the $x$-axis. The dotted figure is the position of $i$ where* meets$(i,j)$ *is maximal.*



meets *for Fuzzy Intervals*

■

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, asymmetry holds, and transitivity does not hold. If $F$ and $S$ widen the peaks then nothing of this can be predicted any more.

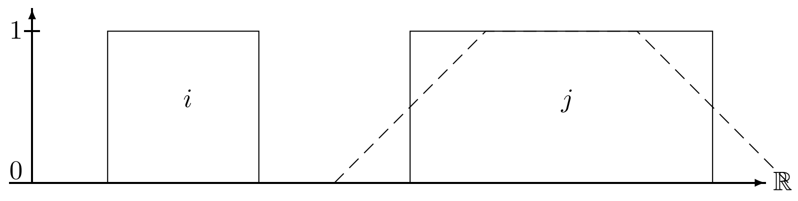
**During for Finite Intervals**

The interval–interval *during*-relation averages the point–interval *during* relation $D_p$ by integrating over the interval $i$. The basic formula is therefore

$$during_{D_p}(i,j) \stackrel{\text{def}}{=} \frac{\int i(x) \cdot D_p(j)(x)\ dx}{|i|}$$

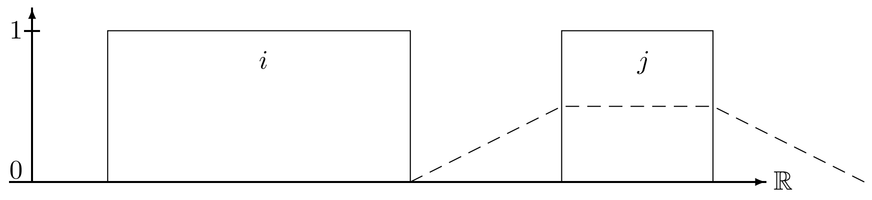
*during*$(i,j)$ measures to what degree $i$ is contained in $j$. The normalization factor is $|i|$ because if $i$ is larger than $j$ then *during*$(i,j)$ should definitely be smaller than 1.

**Example 2.2.62 (*during* for Crisp Intervals)**
*The dashed line in the picture below shows the result of the* during-*relation for a coordinate $x$ when the* middle point *of the interval $i$ is moved to $x$.*



during *for Crisp Intervals*

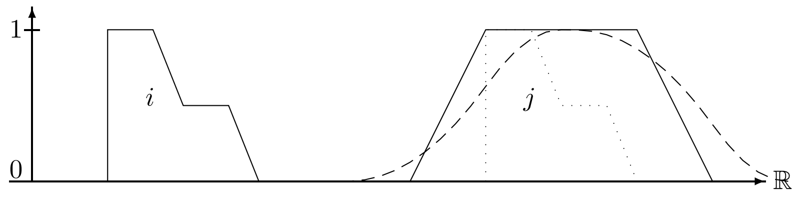
*The interval $i$ in the next picture is larger than $j$ such that* before$(i,j)$ *never rises to 1.*



during *for Crisp Intervals*

■

**Example 2.2.63 (*during* for Fuzzy Intervals)**
*The dashed line shows again the result of the* during-*relation when the middle point of $i$ is moved along the $x$-axis. The dotted figure indicates the position of $i$ where* during$(i,j)$ *is maximal.*



during *for Fuzzy Intervals*

■

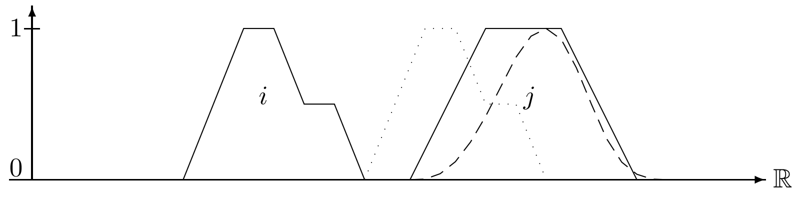
**During for Infinite Intervals**
The formula $\int i(x)\cdot D_p(j)(x)\ dx/|i|$ cannot be evaluated if $i$ is an infinite interval. Instead one can compute $\lim_{a\to\infty}\int_{-a}^{+a} i(x)\cdot D_p(j)(x)\ dx/\int_{-a}^{+a} i(x)\ dx$. If the limes is calculated analytically we obtain $(i(-\infty)\cdot j(-\infty)+i(+\infty)+j(+\infty))/(i(-\infty)\cdot i(+\infty))$. This formula is used for the *during* relation on infinite intervals $i$.

**Overlaps for Finite Intervals**
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 $j$.

The first condition is encoded in the factor $1 - D(i, E^+(j))$ where $D$ is a *during* operator. $E^+(j)$ extends the rising part of $j$ to 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 $\max_a((1 - D(shift_a(i), E^+(j))) \cdot D(shift_a(i), j))$ which corresponds to the maximal possible overlap when $i$ is shifted along the $x$-axis. This guarantees that there is a position for $i$ where *overlaps*$(i, j) = 1$.

**Example 2.2.64 (*overlaps* for Fuzzy Intervals)** *This example shows the result of the* overlaps *relation where the standard* during *operator is used (with the identity function as point–interval* during *operator).*



*Example: Overlaps Relation*

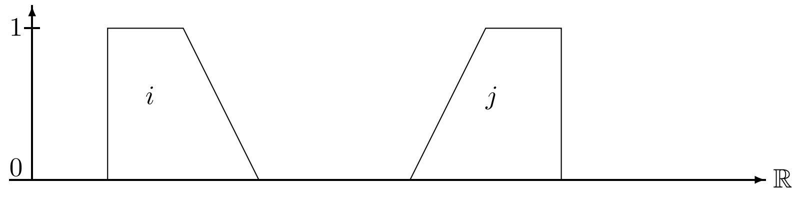
*The dashed line represents the result of the overlaps relation for an $x$-coordinate $x$ where the positive end of the interval $i$ is moved to $x$. The dotted figure indicates the interval $i$ moved to the position where* overlaps$(i, j)$ *becomes maximal.*

■

The normalization factor $\max_a((1 - D(shift_a(i), E^+(j))) \cdot D(shift_a(i), j))$ 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.

**Overlaps for Infinite Intervals**
If the interval $j$ is negative infinite then there cannot be anything before $j$. Therefore *overlaps*$(i, j) = 0$. In the other infinite cases we compute the crisp overlaps relation for the crisp hulls of $i$ and $j$.

**Properties of the *overlaps* relation**: The crisp *overlaps* relation is irreflexive, asymmetric and not transitive. For the standard *overlaps*$_{st}$ relation and finite fuzzy sets $i$ which are crisp at the left side we have $D_{st}(i, E^+(i)) = 1$. Therefore, *overlaps*$_{st}(i, i) = (1 - D_{st}(i, E^+(i)) \cdot D_{st}(i, i)/N'(i, j) = 0$ holds. If $i$ is fuzzy at the left side then $D_{st}(i, E^+(i)) < 1$ and $D_{st}(i, i) > 0$, and therefore *overlaps*$_{st}(i, i) > 0$.



overlaps$_{st}(i, i) = 0$ *and* overlaps$_{st}(j, j) > 0$

For crisp intervals $i$ and $j$ we have a property which corresponds to asymmetry: $\mathit{overlaps}_{st}(i,j) > 0 \Rightarrow \mathit{overlaps}_{st}(j,i) = 0$. This is because $\mathit{overlaps}_{st}(i,j) > 0$ 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 $\mathit{overlaps}_{st}(i,j) > 0$ and $\mathit{overlaps}_{st}(j,i) > 0$.

**Starts** and **Finishes**
The fuzzy *finishes*-relation is the symmetric variant of the fuzzy *starts*-relation. Therefore we need to consider only the *starts*-relation. 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 $j$.

This led to the basic definition of the fuzzy *starts*-relation as a product of the overlap between the two starting sections of $i$ and $j$, and the $\mathit{during}(i,j)$-relation:

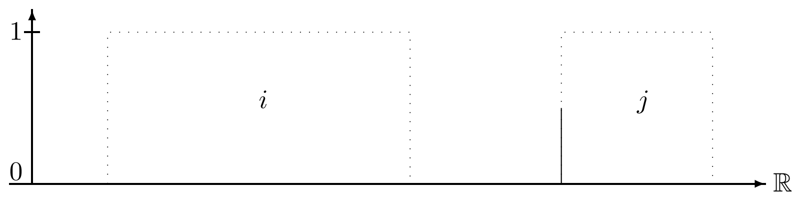
$$\mathit{starts}(i,j) \overset{\text{def}}{=} \frac{\int S_1(i)(x) \cdot S_2(j)(x)\ dx}{N(S_1(i), S_2(j))} \cdot D(i,j)$$

The first factor checks the first condition: the starting part $S_1(i)$ of $i$ should coincide with the starting part $S_2(j)$ of $j$. This value is normalized to the maximal possible overlap of the starting parts. The assumption here is that if I move $i$ along the $\mathbb{R}$-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 $\mathit{starts}(i,j)$ can be $< 1$ regardless of the position of $i$.

The extreme cases are:

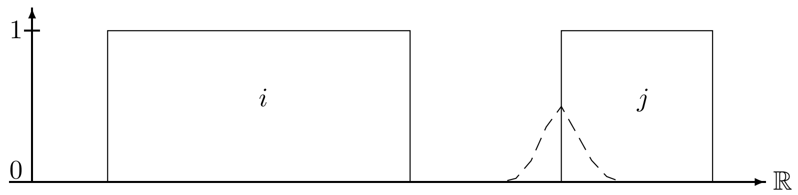
- if $i$ or $j$ are empty then $\mathit{starts}(i,j)$ must be 0;
- if one of $i$ and $j$ is negative infinite then they can't have the same starting point. Therefore $\mathit{starts}(i,j)$ must be 0 again;
- if both are negative infinite then we can assume that they have the same starting point, and therefore only the second condition $\mathit{during}(i,j)$ 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 2.2.65** starts *for Crisp Intervals*
*The first picture shows the case where a* during *operator $D$ is used with $D_d$ = identity. If the front end of $i$ is moved along the $x$-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 $j$.*



*Crisp* starts-*Relation*

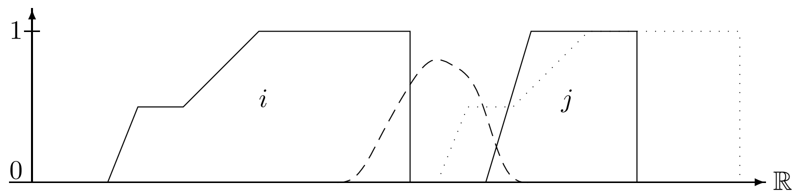
*The next picture shows a fuzzified* starts-*relation. The dashed line shows the value of* starts$(i,j)$ *for a position $x$ where the* front end *of $i$ is moved to $x$. The crisp intervals are fuzzified in the same way as in Example 2.2.60. The peak is broader, but the maximum is still at 0.5 because $i$ is twice as large as $j$.*



*Fuzzified* starts-*relation*

■

**Example 2.2.66 (*starts*-Relation for Fuzzy Intervals)** *The next figure shows the application of the same* starts-*relation as in the first picture of the above example to fuzzy intervals. The dashed line is again the result of the* starts-*relation. The dotted figure shows the position of the interval $i$ where* starts$(i,j)$ *is maximal.*



*Example: Starts Relation*

■

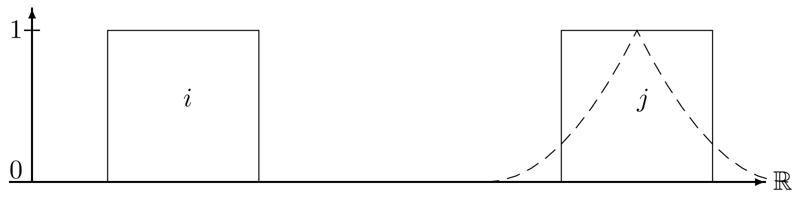
**Properties of the *starts* relation**: The classical starts relation for crisp intervals is reflexive, antisymmetric and transitive. If we consider only the cases $starts_{st}(i,j) = 0$ or $starts_{st}(i,j) = 1$ where $i$ and $j$ are crisp intervals then it behaves like the classical starts relation. The same properties hold, in particular reflexivity. For fuzzy intervals, however, we have $0 < starts_{st}(i,i) < 1$. No kind of fuzzy antisymmetry holds for $starts_{st}$ 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.

**Equals:**
An interval $i$ equals an interval $j$ if $i$ is a subset of $j$ and vice versa. Therefore we get

$$equals_D(i,j) \overset{\text{def}}{=} D(i,j) \cdot D(j,i)$$

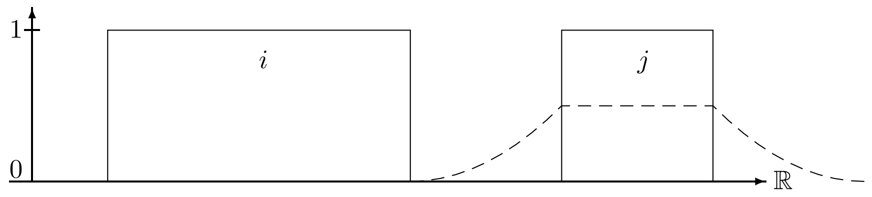
where $D$ is an interval–interval *during* operator.

**Example 2.2.67 (*equals* for Crisp Intervals)**
*The first picture shows the* equals-*relation for similar intervals. If $i$ is moved on top of $j$ then* equals$(i,j) = 1$



equals *for Similar Intervals*

*$i$ and $j$ in the next figure are not equal. Therefore* equals$(i, j)$ *never rises to 1.*



equals *for Different Intervals*

■

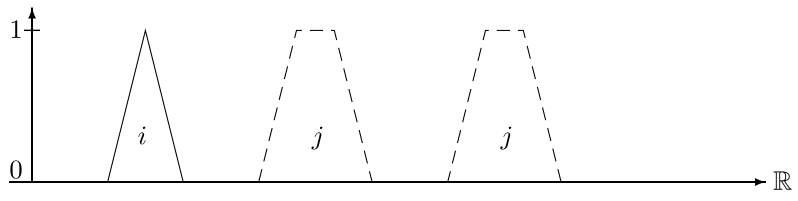
**Properties of the *equals* relation**: The classical equals relation is an equivalence relation. $\mathit{equals}_{st}(i, j) = 1$ implies $i$ and $j$ are crisp and moreover $i = j$ (see the corresponding remark about $\mathit{during}_{st}(i, j) = 1$). Therefore if we consider only the case $\mathit{equals}_{st}(i, j) = 1$ then $\mathit{equals}_{st}$ is also an equivalence relation.

Since $\mathit{during}_{st}(i, i) > 0$ we also have $\mathit{equals}_{st}(i, i) > 0$ for non-empty fuzzy sets. By the very definition of $\mathit{equals}_D$ we have $\mathit{equals}_D(i, j) = \mathit{equals}_D(j, i)$, the strongest form of fuzzy symmetry. Any form of fuzzy transitivity does not hold. It is easy to construct examples where $\mathit{equals}_{st}(i, j) > 0$ and $\mathit{equals}_{st}(j, k) > 0$ and $\mathit{equals}_{st}(i, k) = 0$. This is because although $\mathit{equals}_{st}(i, j) > 0$ requires an overlap between $i$ and $j$, and $\mathit{equals}_{st}(j, k) > 0$ requires an overlap between $j$ and $k$, there need not be an overlap between $i$ and $k$.

**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.

### 2.2.9 The Search Problem

The normalization factor $N(i, j) = \max_a \int i(x-a) \cdot j(x)\, dx$, where $i$ is finite, amounts in general to a nontrivial search problem with unpredictable solutions. Consider the following example:



*Maximizing 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 maximized 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.

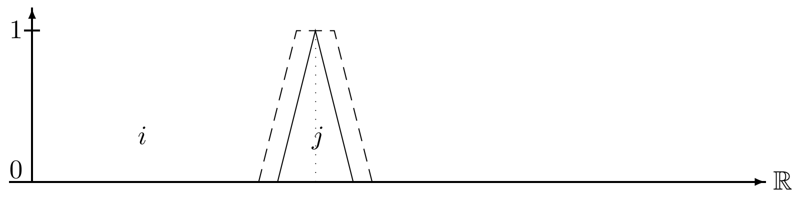
The first class is when $j$ is infinite and $j(-\infty) = 1$ or $j(+\infty) = 1$, and, of course $j$ has a finite kernel. In this case one can move $i$ to the infinite part where $j$ is constant 1. $\int i(x-a)j(x)\ dx = |i|$ in this case, i.e. $\max_a \int i(x-a) \cdot j(x)\ dx = |i|$,

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

**Definition 2.2.68 (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.

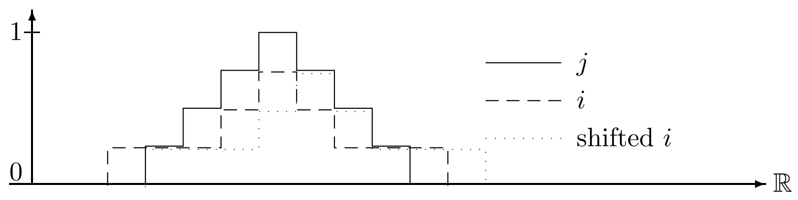


*Maximal Overlap*

**Proposition 2.2.69** *If $i$ and $j$ are two monotone and symmetric fuzzy intervals then $\int_{-\infty}^{+\infty} i(x)j(x)\ dx$ 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 discretesized into step functions with finite step size. The limit 'step size $\mapsto$ 0' is then the original problem. The discretesized 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.

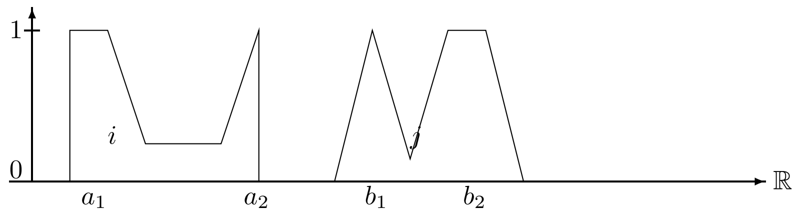


*Discretesized Maximization Problem*

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 $\int_{-\infty}^{+\infty} i(x-a)j(x)\ dx$ 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 FuTIRe is a combination of an iterated binary local search with a randomized global search. It is optimized 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 $\Delta$ 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 $\Delta$ as before. This is iterated until the changes in the integral falls under another threshold (1% seemed to be a good choice).

**Definition 2.2.70 (The Search for Maximizing the Overlap)** *Let $i$ and $j$ be two* finite *fuzzy intervals. We define a local search function and then a global search procedure for maxi-*

*mizing the integral*

$$Int(a) \stackrel{\text{def}}{=} \int_{-\infty}^{+\infty} i(x-a)j(x)\ dx.$$

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

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

*$iteratedLocalSearch(a,\Delta)$: 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.* ■

## 2.3 Nagypál and Motik's Interval–Interval Relations

An alternative approach to fuzzy interval–interval relations has been proposed by Nagypál and Motik [33]. They define interval–interval relations directly, and not via point–interval relations. The basic idea 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.

Since these relations are also implemented in the FuTIRe library, we briefly describe how they work. More details are given in the original paper [33].

**Definition 2.3.1 (Nagypál and Motik's Interval–Interval Relations)** *Let $i$ and $j$ be two fuzzy intervals.*

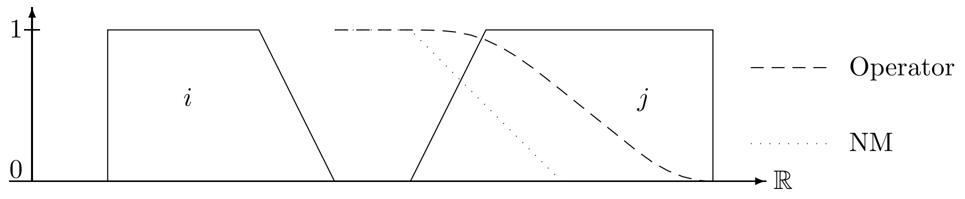
$$
\begin{array}{lll}
\text{before}(i,j) & \overset{\text{def}}{=} & sup_x(((1 - extend^-(i)) \cap_{min} (1 - extend^+(j)))(x)) \\
\text{meets}(i,j) & \overset{\text{def}}{=} & min(inf_x((extend^-(i) \cup_{max} extend^+(j))(x)), \\
 & & \quad inf_x(((1 - extend^-(i)) \cup_{max} (1 - extend^+(j)))(x))) \\
\text{overlaps}(i,j) & \overset{\text{def}}{=} & min(sup_x((extend^+(i) \cap_{min} (1 - extend^+(j)))(x)), \\
 & & \quad sup_x((extend^-(i) \cap_{min} extend^+(j))(x)), \\
 & & \quad sup_x(((1 - extend^-(i)) \cap_{min} extend^-(j))(x))) \\
\text{starts}(i,j) & \overset{\text{def}}{=} & min(inf_x(((1 - extend^+(i)) \cup_{max} extend^+(j))(x)), \\
 & & \quad inf_x((extend^+(i) \cup_{max} (1 - extend^+(j))(x))), \\
 & & \quad sup_x(((1 - extend^-(i)) \cap_{min} extend^-(j))(x))) \\
\text{during}(i,j) & \overset{\text{def}}{=} & min(sup_x(((1 - extend^+(i)) \cap_{min} extend^+(j))(x)), \\
 & & \quad sup_x(((1 - extend^-(i)) \cap_{min} extend^-(j))(x))) \\
\text{finishes}(i,j) & \overset{\text{def}}{=} & min(inf_x((extend^-(i) \cup_{max} (1 - extend^-(j))(x))), \\
 & & \quad inf_x(((1 - extend^-(i)) \cup_{max} extend^-(j))(x)), \\
 & & \quad sup_x((extend^+(i) \cap_{min} (1 - extend^+(j)))(x))) \\
\text{equals}(i,j) & \overset{\text{def}}{=} & min(inf_x((extend^-(i) \cup_{max} (1 - extend^-(j))(x))), \\
 & & \quad inf_x(((1 - extend^-(i)) \cup_{max} extend^-(j))(x)), \\
 & & \quad inf_x(((1 - extend^+(i)) \cup_{max} extend^+(j)))(x)) \\
 & & \quad inf_x((extend^+(i) \cup_{max} (1 - extend^+(j)))(x)).
\end{array}
$$

■

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. In contrast to the operator based definitions, the Nagypál and Motik relations behave on crisp intervals just like the classical interval–interval relations.

In the next examples we compare the operator version of the interval–interval relations with the Nagypál and Motik version. The operator version uses always the default parameter setting. Because the Nagypál and Motik version is designed mainly for finite intervals consisting of one component only, we present only examples of this kind.

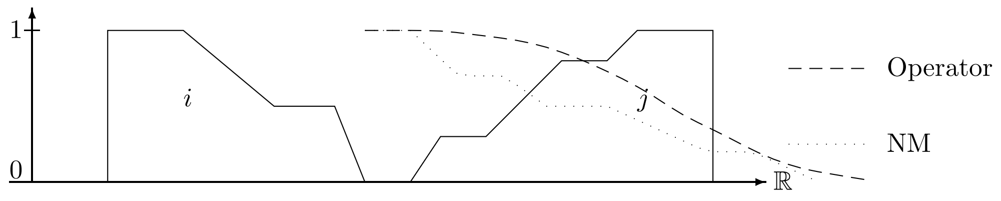
**before**:
The first picture shows the result of the *before* relation when the interval $i$ is moved along the $x$-axis. A point $(x, y)$ at the dashed and dotted curves is the result of the *before* relation when the positive end of the interval $i$ is moved to position $x$.



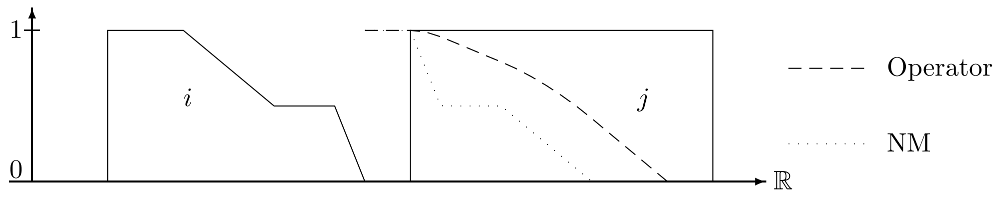
before *relation: standard operator version and Nagypál and Motik version*

In this example the operator version has non-zero fuzzy values 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 picture shows that the Nagypál and Motik version can have a similar effect.



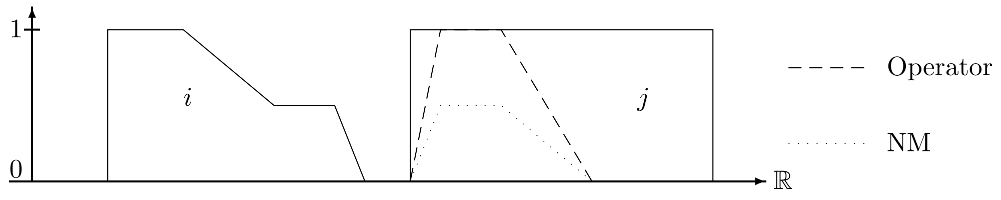
before *relation: standard operator version and Nagypál and Motik version*



before *relation: standard operator version and Nagypál and Motik version*

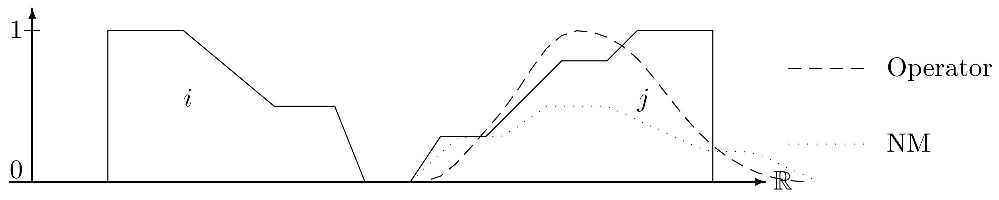
The examples demonstrate that the Nagypál and Motik version of the *before* relation reveals more about the fine structure of the back end of the first interval $i$ and the front end of the second interval $j$, but the precise relation is not very clear. The operator version, on the other hand, has a more global meaning: the resulting fuzzy value stands for fraction of the first interval which is before the second interval.

**meets**:
The next two figures illustrate that the differences between the operator version of the *meets* relation and the Nagypál and Motik version are minor. The operator version yields a smoother curve, and the fuzzy values are normalized 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.



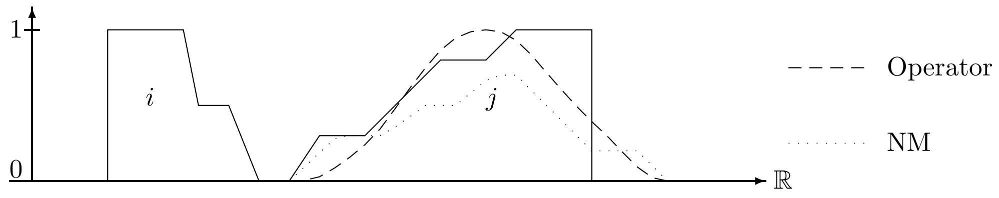
meets *relation: standard operator version and Nagypál and Motik version*

The dashed and dotted curves indicate again the resulting value of the *meets* relation at coordinate $x$ if the interval $i$ is moved such that its positive end is at $x$.



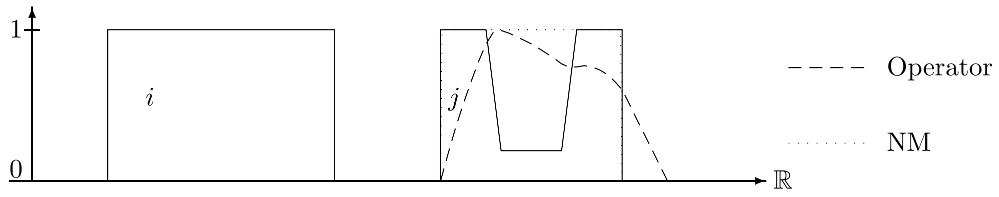
meets *relation: standard operator version and Nagypál and Motik version*

**overlaps**:
The first example for the *overlaps* relation shows a structural similarity between the operator version and the Nagypál and Motik version. Again, the Nagypál and Motik version is not normalized.



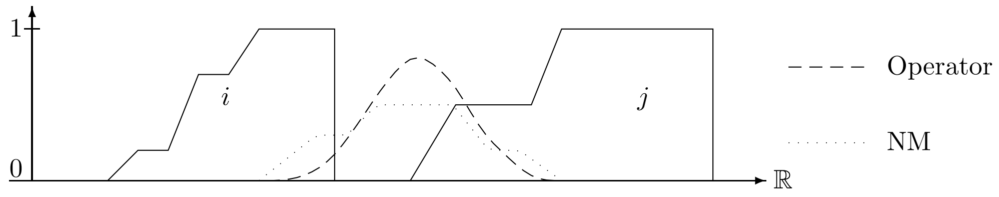
overlaps *relation: standard operator version and Nagypál and Motik version*

The next example demonstrates that the Nagypál and Motik version of the *overlaps* relation is not sensitive to the internal structure of the intervals. The interval $j$ is treated like its crisp hull.



overlaps *relation: standard operator version and Nagypál and Motik version*

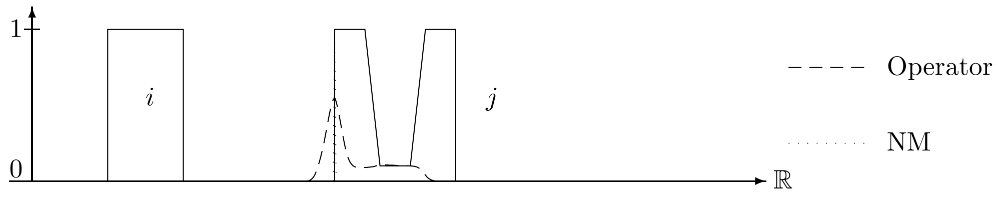
***starts***:
The operator version and the Nagypál and Motik version of the *starts* relation show close structural similarities if the left sides of the intervals are not too exotic.



starts *relation: standard operator version and Nagypál and Motik version*

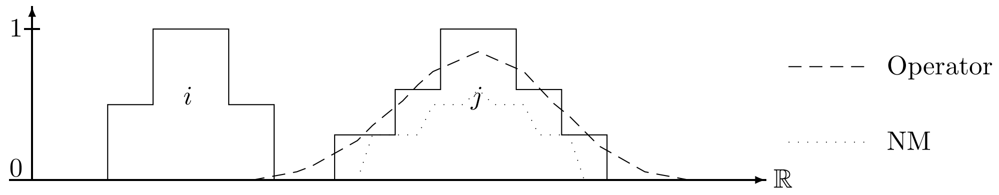
The dashed and dotted lines in this picture show the resulting values of *starts* relation at a point $x$ if the *left* end of the interval $i$ is moved to $x$.

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.



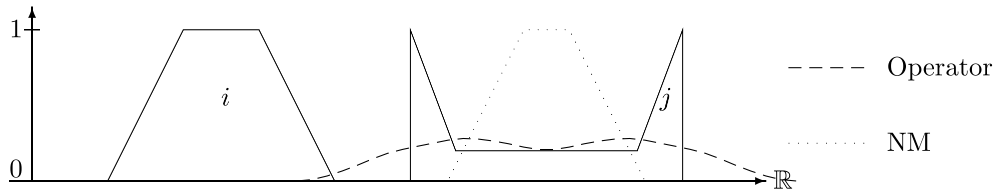
starts *relation: standard operator version and Nagypál and Motik version*

**during**:
The next two pictures show a comparison of the *during* relation. The dashed and dotted lines indicate the values of the during relation at $x$-coordinate $x$ if the middle point of the interval $i$ is moved to $x$. Compared to the operator version, the Nagypál and Motik version has a much narrower graph and it is less smooth.



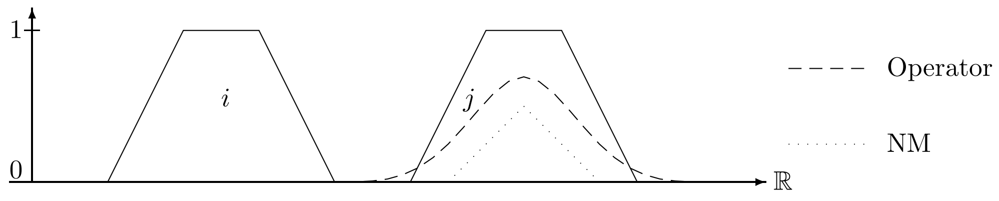
starts *relation: standard operator version and Nagypál and Motik version*

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.



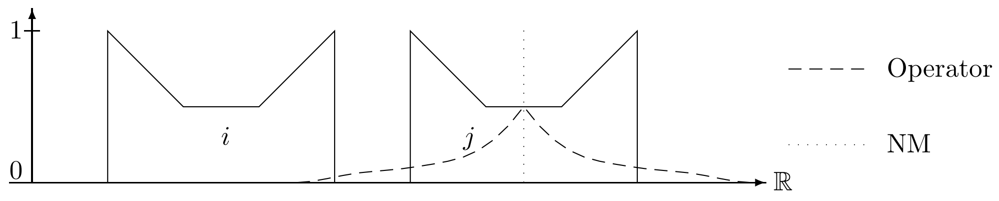
during *relation: standard operator version and Nagypál and Motik version*

**equals**:
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 the rely on the meaning of the *during* relation.



equals *relation: standard operator version and Nagypál and Motik version*

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.



equals *relation: standard operator version and Nagypál and Motik version*

**Summary of the Comparison**:
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 version are more obvious for crisp intervals, where the Nagypál and Motik versions deliberately behave like the pure crisp relations. The operator versions 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.

Since the FuTIRe library offers both versions of the relations, the application has to choose the appropriate version.

## 2.4 Datastructures and Algorithms

The algorithms presented in this document need to deal with five basic datatypes: time points, fuzzy values, fuzzy temporal intervals, y-functions and interval operators.

**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 Cesium 133 atom. Therefore the most precise time measurement available at all depends on counting integers (cycles of light).

Therefore the FuTIRe-library *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 Cesium 133 light.

**Fuzzy Values**
Fuzzy values usually are 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 I decided again to use 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 2.4.1 (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 chapter. $\top$ 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 [45, 24]; 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 2.4.2 ($x$-Integers and $y$-Integers)** *FuTIRe may use 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 library 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 modularized and easy to understand implementations, but it also makes the mathematical notation clearer. Therefore we shall use the notion $o.v$ 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 $\min(i \geq 0 \mid \varphi(i))$ denotes a loop: starting with $i = 0$, increase $i$ by 1 until $\varphi(i)$ becomes true. In this case return the $i$ with $\varphi(i) = \mathit{true}$. Notice that the loop may in general not terminate. We use similar expressions with the obvious meaning.

The expression

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

is a case analysis. It means:
$a \overset{\text{def}}{=} s_1$ if $\varphi_1$ is true
$a \overset{\text{def}}{=} s_2$ if $\varphi_1$ is false and $\varphi_2$ is true
$\ldots$
$a \overset{\text{def}}{=} s_n$ 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*'.

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

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.

## 2.4.1 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. 2.4.2).

**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 [45]. 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 2.4.3 (Collinear)** *The method $p_1.colinear(p_2, p_3)$ 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 2.4.4 (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.* ■

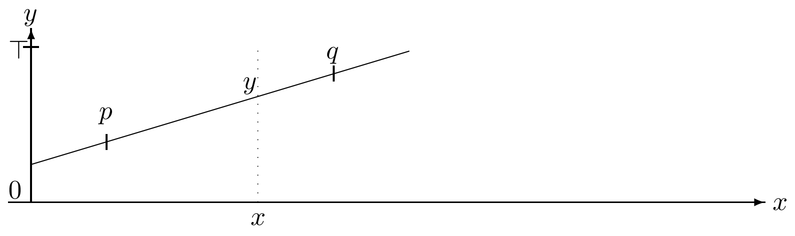
The *$leftturn$* method computes the doubled area of the triangle $p_1, p_2$ and $p_3$ and checks its sign. Left turns and right turns yield opposite signs.

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

**Definition 2.4.5 (IntersectsProper)** *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 2.4.6 (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 2.4.7 (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 2.4.8 (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 2.4.9 (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.*

$$p.Area2(q, x_1, x_2) \overset{\text{def}}{=} \begin{cases} \mathit{undefined} & \text{if } p.x = q.x \text{ and } p.x \neq x \\ 0 & \text{if } p.x = q.x = x \\ (x_2 - x_1) \cdot (p.LineY(q, x_2) - p.LineY(q, x_1)) & \text{otherwise} \end{cases}$$

*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 2.4.10 (Area2X)** *Let $p$ and $q$ be the two points which define a line. Let $a \geq 0$ be an integer or floating point number.*

$$p.Area2X(q, a) \overset{\text{def}}{=} \begin{cases} undefined & \text{if } p.x = q.x \\ & \text{or } p.y = q.y = 0 \text{ and } a > 0 \\ & \text{or } p.y^2 < -slope \cdot a \\ p.x & \text{if } p.y = q.y = 0 \text{ and } a = 0 \\ p.x + roundX(\frac{a}{2p.y}) & \text{if } p.y = q.y \\ p.x + roundX(\frac{\sqrt{p.y^2 + slope \cdot a} - p.y}{slope}) & \text{otherwise} \\ \quad \text{where } slope \overset{\text{def}}{=} \frac{q.y - p.y}{q.x - p.x} & \end{cases}$$

*The result is a rounded $x$-integer.* ■

**Proposition 2.4.11 (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 normalized 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**

The Interval–interval relations (Section 2.4.2) are defined as an integral over two multiplied polygons. A building block for the integration algorithm is a method which integrates the product of two lines.

**Definition 2.4.12 (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.*

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

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

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

*The result is a floating point number.* ■

**Proposition 2.4.13 (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:**

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

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

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

### 2.4.2 Fuzzy Time Intervals

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

**Definition 2.4.14 (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 bitsize 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.

#### Representation and Construction

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

**Definition 2.4.15 (Envelope Polygon)** *The* envelope polygon $I$ *of a fuzzy time interval is a finite sequence of points $p_0, \ldots, p_n$ such that $p_i.x \leq p_{i+1}.x$ holds for all $i$.*

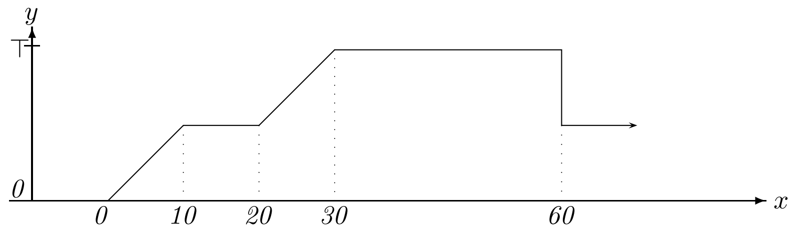
*In most cases we can assume that the coordinates in $I$ are not redundant, i.e. there are no collinear triples $(p_i, p_{i+1}, p_{i+2})$ of points.*

*We usually identify the envelope polygon with the fuzzy set itself.* ■

**Example 2.4.16 (Envelope Polygon)** *The picture below shows the envelope polygon*

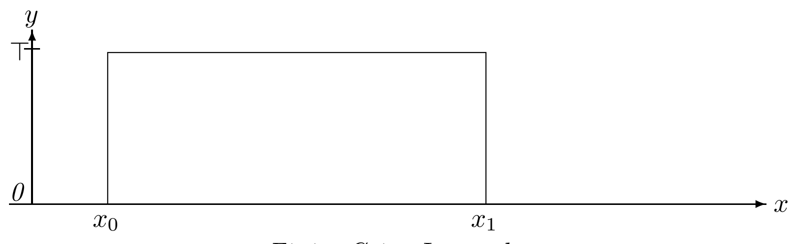
$$I = (0,0)(10,500)(20,500)(30,1000)(60,1000)(60,500).$$

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



*The Envelope Polygon*

**Example 2.4.17 (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*

*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 $p_i.x \leq p_{i+1}.x$ holds and that collinear triples of points are avoided.

**Definition 2.4.18 (push_back and pop_back)** *Let* $I = (p_0, \ldots, p_n)$ *be an envelope polygon and* $p$ *a new point.*

$$I.push\_back(p) \overset{\text{def}}{=} \begin{cases} undefined & \textit{if } I \neq () \textit{ and } p.x < p_n.x \\ (p) & \textit{if } I = () \textit{ or } I = (p_0) \textit{ and } p.y = p_0.y \\ (p_0, p) & \textit{if } I = (p_0) \\ (p_1, \ldots, p_{n-1}, p) & \textit{if } p.colinear(p_{n-1}, p_n) = true \textit{ (Def. 2.4.3)} \\ (p_1, \ldots, p_n, p) & \textit{otherwise} \end{cases}$$

*I.pop_back*() *removes the last element.* ∎

The method *I.close*() defined next 'closes' a polygon in the sense that it assumes that all points have been added with the *push_back*-method and some further redundancies can be removed. For some of the other algorithms it is important that these redundancies are in fact removed.

**Definition 2.4.19 (Close)** *Let* $I = (p_0, \ldots, p_n)$ *be an envelope polygon.*

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

∎

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 $x$. $MaxIndex(true)$ locates the index of the leftmost polygon point with maximum $y$-value ($I^{fm}$), whereas $MaxIndex(false)$ locates the index of $I^{lm}$.

**Definition 2.4.20 (Index and MaxIndex)** *For an envelope polygon* $I = (p_0, \ldots, p_n)$ *let*

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

*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.*

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

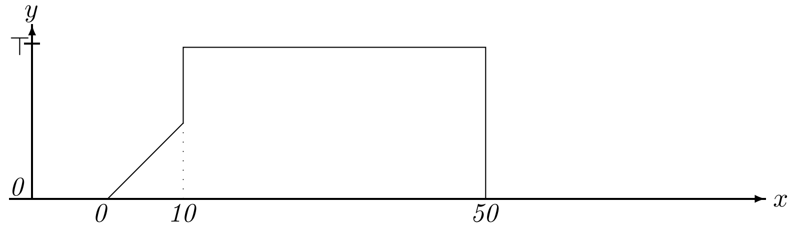
■

*MaxIndex* 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 2.4.21 (Index and MaxIndex)** *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*
*$I.Index(0) = 0$, $I.Index(9) = 0$, $I.Index(10) = 2$, $I.Index(11) = 2$, $I.Index(50) = 4$,*
*$I.MaxIndex(true) = 2$, $I.MaxIndex(false) = 3$.*



*Index and MaxIndex*

■

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

$$I.Member(x) \overset{\text{def}}{=} \begin{cases} 0 & \textit{if } I = () \\ p_0.y & \textit{if } x < x_0 \\ p_n.y & \textit{if } x \geq x_n \\ p_i.y & \textit{if } p.x = p_i.x \\ p_i.lineY(p_{i+1}, x) & \textit{otherwise} \\ \quad \textit{where } i = I.Index(x) & \end{cases}$$

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

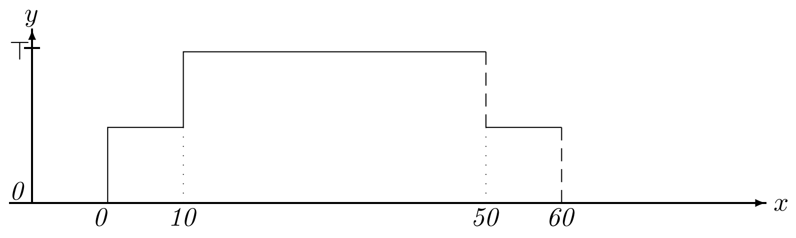
*The usual membership function (Def. 2.2.2) is then $I(x) = I.Member(I, x)/\top$* ■

**Remark 2.4.23 (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 2.4.24 (Half-open Intervals)** *The Index method (Def. 2.4.20) 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 $I.Member(0) = 500$, $I.Member(10) = 1000$, $I.Member(50) = 500$, $I.Member(60) = 0$.* ■

**Remark 2.4.25 (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;*
- *$I = ((a,y))$ 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*

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

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

■

**Basic Features of Fuzzy Intervals**

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

**Definition 2.4.26 (Infinity Predicates)** *Let $I = (p_0,\ldots,p_n)$ be an envelope polygon.*

$I.isNegInfinite() \overset{\text{def}}{=} I \neq ()$ *and* $p_0.y > 0$
$I.isPosInfinite() \overset{\text{def}}{=} I \neq ()$ *and* $p_n.y > 0$
$I.isInfinite() \overset{\text{def}}{=} I \neq ()$ *and* $p_0.y > 0$ *or* $p_n.y > 0$

■

Using the $MaxIndex$-method (Def. 2.4.20) we can define $I.Max()$ for computing the height $I\hat{}$ (Def. 2.2.10) of the fuzzy interval. We also define $MaxX(front)$ to compute the $x$-coordinate of the first maximal point $i^{fm}$ (Def. 2.2.11) if $front = true$ and $MaxX(false)$ which computes the last maximum $i^{lm}$ (Def. 2.2.11).

**Definition 2.4.27 (Min and Max Values)** *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon.*

$$I.Min() \overset{\text{def}}{=} min_{0 \leq i \leq n} p_i.y$$

$$I.Max() \overset{\text{def}}{=} \begin{cases} 0 & \textit{if } I = () \\ p_{I.MaxIndex(true)}.y & \textit{otherwise} \end{cases}$$

$$I.MaxX(true) \overset{\text{def}}{=} \begin{cases} -\infty & \textit{if } I = () \textit{ or } I.MaxIndex(true) = 0 \textit{ and } p_0.y > 0 \\ p_{I.MaxIndex(true)}.x & \textit{otherwise} \end{cases}$$

$$I.MaxX(false) \overset{\text{def}}{=} \begin{cases} +\infty & \textit{if } I = () \textit{ or } I.MaxIndex(false) = n \textit{ and } p_n.y > 0 \\ p_{I.MaxIndex(false)}.x & \textit{otherwise} \end{cases}$$

*The result of $Max$ is an $y$-integer value and the result of $MaxX$ is an $x$-integer value.* ■

The complexity of $Max$ and $MaxX$ is in general linear because $MaxIndex$ requires linear search. It is constant for crisp intervals.

**Proposition 2.4.28 (Soundness of Max and MaxX)** *For an envelop polygon $I$ we have $I.Max()/\top = I\hat{}$, $I.MaxX(true) = I^{fm}$ and $I.MaxX(false) = I^{lm}$.*

*The proofs are straightforward.* ■

**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. $I.Size()/\top = 2|I|$. $I.Size2(k,l)$ 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 2.4.29 (Size)** *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon. Let $k$ and $l$ be two indices.*

$$I.Size2I(k,l) \overset{\text{def}}{=} \begin{cases} undefined & \textit{if } k < 0 \textit{ or } l > n \\ -I.Size2(l,k) & \textit{if } l < k \\ 0 & \textit{if } k = l \textit{ or } I = () \\ \Sigma_{m=k}^{l-1} p_m.Area2(p_{m+1}) & \textit{otherwise} \quad \textit{(Def.2.4.9)} \end{cases}$$

$$I.Size2() \overset{\text{def}}{=} \begin{cases} 0 & \textit{if } I = () \\ +\infty & \textit{if } p_0.y > 0 \textit{ or } p_n.y > 0 \\ I.Size2(0,n) & \textit{otherwise} \end{cases}$$

*Both versions of $Size2$ return $x$-integers.*

*Now let $a$ and $b$ be two $x$-coordinates:*

$$I.Size2(a,b) \overset{\text{def}}{=} \begin{cases} 0 & \textit{if } I = () \textit{ or } a = b \\ -I.Size2(b,a) & \textit{if } b < a \\ 2 \cdot (b-a) \cdot p_n.y & \textit{if } a \geq p_n.x \\ 2 \cdot (b-a) \cdot p_0.y & \textit{if } b \leq p_0.x \\ (b-a) \cdot (p_i.LineY(p_{i+1}, a) + p_i.LineY(p_{i+1}, b)) & \textit{if } p_{i-1}.x \leq a \leq b \leq p_i.x \\ \quad \textit{where } i = I.Index(a) & \\ head + middle + tail & \textit{otherwise} \end{cases}$$

*where*

$$head \overset{\text{def}}{=} \begin{cases} 2 \cdot (p_0.x - a) \cdot p_0.y & \textit{if } a \leq p_0.x \\ 0 & \textit{if } p_i.x = a \\ p_i.Area2(p_{i+1}, a, p_{i+1}.x) & \textit{otherwise} \\ \quad \textit{where } i = I.Index(a) \textit{ and} & \end{cases}$$

$$middle \overset{\text{def}}{=} I.Size2(I.Index(a), I.Index(b)) \textit{ and}$$

$$tail \overset{\text{def}}{=} \begin{cases} 2 \cdot (b - p_n.x) \cdot p_n.y & \textit{if } b \geq p_n.x \\ 0 & \textit{if } p_i.x = b \\ p_i.Area2(p_{i+1}, p_i.x, b) & \textit{otherwise} \\ \quad \textit{where } i = I.Index(b) & \end{cases}$$

*The method returns a floating point value.* ■

The next two methods compute the center and middle points for a fuzzy interval (Def. 2.2.12).

**Definition 2.4.30 (Center Points)** *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon.*

$$I.CenterPoint(k,m) \overset{\text{def}}{=} \begin{cases} undefined & \textit{if } I = () \textit{ or } I.isInfinite() \\ p_0.x & \textit{if } k = 0 \\ p_n.x & \textit{if } k = m \\ p_{i-1}.Area2X(p_i, \frac{s2 \cdot k}{m} - I.Size2(0, i-1)) & \textit{otherwise} \\ \quad \textit{where } s2 \overset{\text{def}}{=} I.Size2(0,n) \textit{ and} & \\ \quad i = \min(i \mid m \cdot I.Size2(0,i) > s2 \cdot k) & \end{cases}$$

*$I.MiddlePoint(n,m) \overset{\text{def}}{=} I.CenterPoint(2n+1, 2m)$.*

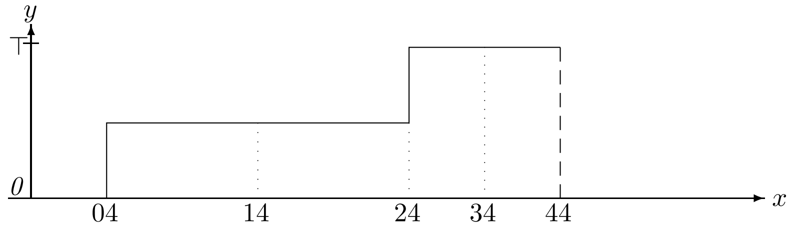
*Both functions return a (rounded) $x$-integer.*

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

The CenterPoint 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 2.4.31 (Center Point Computation)**

*Let* $I = \underbrace{(0,0)}_{p_0}\underbrace{(0,500)}_{p_1}\underbrace{(4,500)}_{p_2}\underbrace{(4,1000)}_{p_3}\underbrace{(6,1000)}_{p_4}\underbrace{(6,0)}_{p_5}$



*Center Point Computation*

*We have $|I| = 4000$, i.e. $s2 = 8000$, and we want to compute $CenterPoint(1,4)$. 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 2.4.32 (Number of Components)** *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon.*

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

■

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

**Definition 2.4.33 (Component)** *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon.*

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

*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
```

■

**Core, Support and Kernel (Def. 2.2.4)**
The FuTIRe-library provides for each of these three concepts the following 5 methods:

1. $Size()$ measures the size in $x$-coordinates;
2. $List()$ yields an ordered list $((i_0, j_0), \ldots)$ of indices of start and endpoints of the components;
3. $Crisp()$ yields a new envelope polygon with the crisp versions of the components.
4. $First()$ returns the first $x$-coordinate of the concept
5. $Last()$ returns the last $x$-coordinate of the concept.

$Size()$, $First()$ and $Last()$ return $x$-coordinates, $Crisp()$ returns an envelope polygon and $List()$ returns a list of index pairs.

**Definition 2.4.34 (Algorithms for the Core)** *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon.*

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

$$I.CList() \overset{\text{def}}{=} \begin{cases} () \quad \textit{if } I = () & \\ V_{i=0}^{n} \begin{cases} ((0,0)) & \textit{if } i = 0 \textit{ and } p_0.y = \top \\ ((n,n)) & \textit{if } i = n \textit{ and } p_n.y = \top \\ (i, i+1) & \textit{if } i < n \textit{ and } p_i.y = p_{i+1}.y = \top \\ () & \textit{otherwise} \end{cases} & \textit{otherwise} \end{cases}$$

$$I.CCrisp() \overset{\text{def}}{=} \begin{cases} () \quad \textit{if } I = () \\ V_{i=0}^{n} \begin{cases} ((p_0.x, \top)(p_0.x, 0)) & \textit{if } i = 0 \textit{ and } p_0.y = \top \\ ((p_n.x, 0)(p_n.x, \top)) & \textit{if } i = n \textit{ and } p_n.y = \top \\ ((p_i.x, 0)(p_i.x, \top)(p_{i+1}.x, \top)(p_{i+1}.x, 0)) & \textit{if } i < n \textit{ and } p_i.y = p_{i+1}.y = \top \\ () & \textit{otherwise} \end{cases} \end{cases}$$

$$I.CFirst() \overset{\text{def}}{=} \begin{cases} undefined & \textit{if } I = () \\ -\infty & \textit{if } p_0.y = \top \\ \min(p_i.x \mid p_i.y = \top) & \textit{if this is defined} \\ undefined & \textit{otherwise} \end{cases}$$

$$I.CLast() \overset{\text{def}}{=} \begin{cases} undefined & \textit{if } I = () \\ +\infty & \textit{if } p_n.y = \top \\ \max(p_i.x \mid p_i.y = \top) & \textit{if this is defined} \\ undefined & \textit{otherwise} \end{cases}$$

■

**Definition 2.4.35 (Algorithms for the Support)** *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon.*

$$I.SSize() \overset{\text{def}}{=} \begin{cases} 0 & \textit{if } I = () \\ +\infty & \textit{if } I.isInfinite() \\ \Sigma_{i=0}^{n-1}(p_{i+1}.x - p_i.x) & \textit{if } p_i.y > 0 \textit{ or } p_{i+1}.y > 0 \\ 0 & \textit{otherwise} \end{cases}$$

$$I.SList() \overset{\text{def}}{=} \begin{cases} () \quad \textit{if } I = () & \\ (V_{i=1}^{n-1} \begin{cases} ((s,i)) \textit{ and } s := i+1 & \textit{if } p_i.y = p_{i+1}.y = 0 \textit{ and } p_{i-1}.y > 0 \\ () & \textit{otherwise} \end{cases} & \\ (s,n)) & \textit{otherwise} \\ \textit{where } s := 0 \textit{ initially} & \end{cases}$$

$$I.SCrisp() \overset{\text{def}}{=} \begin{cases} () & \textit{if } I = () \\ \left( \begin{array}{l} \begin{cases} ((p_0.x, 0)(p_0.x, \top)) & \textit{if } p_0.y = 0 \\ ((p_0.x, \top)) & \textit{otherwise} \end{cases}, \\ V_{i=1}^{n-1} \begin{cases} ((p_i.x, \top)(p_i.x, 0)) & \textit{if } p_i.y = p_{i+1}.y = 0 \textit{ and } p_{i-1}.y > 0 \\ ((p_i.x, 0)(p_i.x, \top)) & \textit{if } p_i.y = p_{i-1}.y = 0 \textit{ and } p_{i+1}.y > 0 \\ () & \textit{otherwise} \end{cases}, \\ \begin{cases} ((p_n.x, \top)(p_n.x, 0)) & \textit{if } p_n.y = 0 \\ ((p_n.x, \top)) & \textit{otherwise} \end{cases} \end{array} \right) \\ \textit{otherwise} \end{cases}$$

$$I.SFirst() \overset{\text{def}}{=} \begin{cases} undefined & \textit{if } I = () \\ -\infty & \textit{if } p_0.y > 0 \\ p_0.x & \textit{otherwise} \end{cases}$$

$$I.SLast() \overset{\text{def}}{=} \begin{cases} undefined & \textit{if } I = () \\ +\infty & \textit{if } p_n.y > 0 \\ p_n.x & \textit{otherwise} \end{cases}$$

■

**Definition 2.4.36 (Algorithms for the Kernel)** *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon.*

$$I.KSize() \overset{\text{def}}{=} \begin{cases} 0 & \textit{if } I = () \\ p_n.x - p_0.x & \textit{otherwise} \end{cases}$$

$$I.KList() \overset{\text{def}}{=} \begin{cases} () & \textit{if } I = () \textit{ or } p_0.x = p_n.x \\ (0, n) & \textit{otherwise} \end{cases}$$

$$I.KCrisp() \overset{\text{def}}{=} \begin{cases} () & \textit{if } I = () \textit{ or } p_0.x = p_n.x; \\ ((p_0.x, 0)(p_0.x, \top)(p_n.x, \top)(p_n.x, 0)) & \textit{otherwise} \end{cases}$$

$$I.KFirst() \overset{\text{def}}{=} \begin{cases} undefined & \textit{if } I = () \\ p_0.x & \textit{otherwise} \end{cases}$$

$$I.KLast() \overset{\text{def}}{=} \begin{cases} undefined & \textit{if } I = () \\ p_n.x & \textit{otherwise} \end{cases}$$

■

### Hull Operators

The method *I.CrispHull*() implements the *CrH*()-function (Def. 2.2.30).

**Definition 2.4.37 (Crisp Hull)** *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon.*

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

■

The method $I.MonotoneHull()$ implements the $MoH()$-function (Def. 2.2.32). 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 2.4.38 (Monotone Hull)** *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon. We describe the algorithm $I.MonotoneHull()$ procedurally:*

```
If $I = ()$ return ();
Let $newI_1 \overset{\text{def}}{=} (p_0, V_{i=0}^{I.FirstMax()} \begin{cases} ((p_{i-1}.LineX(p_i, max), max), p_i) \\ \quad \text{if } i > 0 \text{ and } p_i.y \geq max \text{ and } p_{i-1}.y < max \\ (p_i) & \text{if } p_i.y \geq max \\ () & \text{otherwise} \end{cases})$
where max is the current largest y-coordinate in $newI_1$:
Let $newI_2 \overset{\text{def}}{=} (p_n, V_{i=n}^{I.LastMax()} \begin{cases} ((p_{i-1}.LineX(p_i, max), max), p_i) \\ \quad \text{if } i < n \text{ and } p_i.y \geq max \text{ and } p_{i-1}.y < max \\ (p_i) & \text{if } p_i.y \geq max \\ () & \text{otherwise} \end{cases})$
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 $newI_1$.
```
■

Finally we implement the convex hull function $CoH$ (Def. 2.2.31). The algorithm 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 2.4.39 (Convex Hull)** *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon. We describe the algorithm $I.ConvexHull()$ procedurally:*

```
If $I = ()$ return ().
Let $i_f \overset{\text{def}}{=} \begin{cases} I.FirstMax() & \text{if } p_0.y > 0 \\ 0 & \text{otherwise} \end{cases}$
Let $i_l \overset{\text{def}}{=} \begin{cases} I.LastMax() & \text{if } p_n.y > 0 \\ n & \text{otherwise} \end{cases}$
Let $newI \overset{\text{def}}{=} (p_{i_f})$.
$For_{i=i_f}^{i_l}$   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.
```
■

### Basic Unary Transformations

A number of basic unary transformations (Def. 2.2.33) can be implemented by just manipulating the vertices of the envelope polygons.

**Definition 2.4.40 (Extend, Scaleup, Shift)** *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon. The three functions return $()$ if $I = ()$. Let $M \overset{\text{def}}{=} (q_0, \ldots, q_k) \overset{\text{def}}{=} I.MontoneHull()$.*

$$
\begin{array}{lll}
I.Extend(true) & \overset{\text{def}}{=} & (V_{i=0}^{j}(q_i), (q_j.x, \top)) \\
 & & where\ j = I.MaxIndex(true) \\
I.Extend(false) & \overset{\text{def}}{=} & ((q_j.x, \top), V_{i=j}^{k}(q_i)) \\
 & & where\ j = I.MaxIndex(false) \\
I.ScaleUp() & \overset{\text{def}}{=} & V_{i=0}^{n}(p_i.x, roundY((p_i.y \cdot \top / I.Max()))) \\
I.Shift(a) & \overset{\text{def}}{=} & V_{i=0}^{n}(p_i.x + a, p_i.y)
\end{array}
$$

■

$Extend(true)$ implements $extend^+$, $Extend(false)$ implements $extend^-$, *ScaleUp* implements *scaleup* and $Shift$ implements $shift$ (Def. 2.2.33). *ScaleUpD* is a destructive version of *ScaleUp* and *ShiftD* is a destructive version of $Shift$.

**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 2.4.41 (Cut)** *Let $I = (p_0, \dots, p_n)$ be an envelope polygon. $x$, $x_1$ and $x_2$ are $x$-coordinates.*

$$
I.Cut(x_1, x_2) \overset{\text{def}}{=} \begin{cases} () & if\ x_2 \le x_1 \\ ((x_1, 0), (x_1, I.Member(x_1)), (V_{i=I.Index(x_1)}^{I.Index(x_2)} p_i), (x_2, I.Member(x_2)), (x_2, 0)) \\ \quad otherwise \end{cases}
$$

*where the list is formed with the push_back operator (Def. 2.4.18). This removes certain redundancies.*

*Let $i_1$ and $i_2$ be two indices.*

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

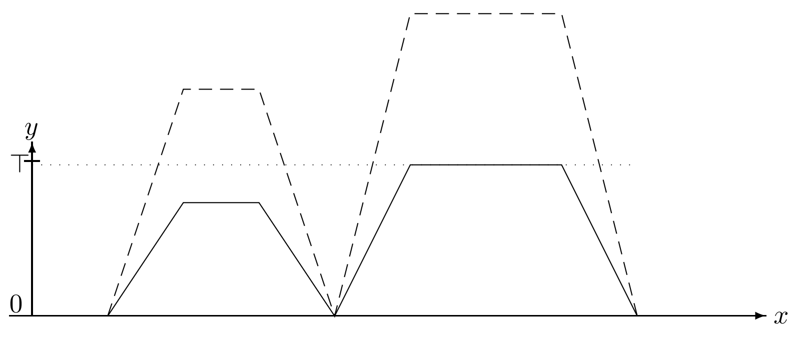
$$
\begin{array}{lll}
I.Cut(x, true) & \overset{\text{def}}{=} & ((x, 0), (x, roundY(I.Member(x))), V_{i=I.Index(x)}^{n} p_i) \\
I.Cut(x, false) & \overset{\text{def}}{=} & (V_0^{i=I.Index(x)} p_i, (x, roundY(I.Member(x))), (x, 0))
\end{array}
$$

■

**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 2.4.42 (Times)** *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon and $a$ a non-negative floating point number.*

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

■

**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.

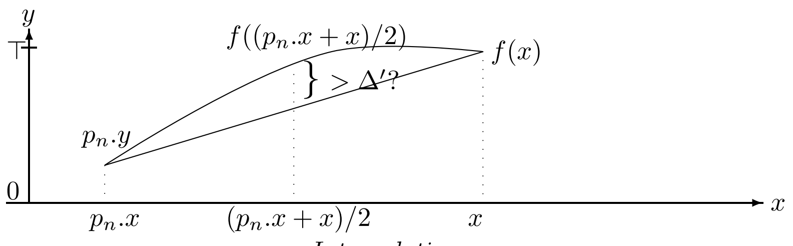
**Definition 2.4.43 (Interpolation)** *Let $I = (p_0, \ldots, p_n)$ be a non-empty envelope polygon, $x$ an x-coordinate, $f$ a function from x-coordinate $\mapsto$ y-coordinate and $\Delta$ a threshold value (e.g. $\Delta = 0.1$).*

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

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

■

The *Interpolate*-method starts with an envelope polygon $I = ((x_0, y_0))$ and fills up $I$ with interpolated values. Suppose $I = (p_0, \ldots, p_n)$. 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 $\Delta$. 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 $\Delta$ is only a basic threshold for very small $y$-values. The threshold $\Delta$' causes that the interpolation becomes denser for larger $y$-values.



*Interpolation*

**Integration**
The *integrate*$^+$-function (Def. 2.2.33) 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. 2.4.9). *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 2.4.44 (Integration)** *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon and $\Delta$ 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.
```
■

## 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 2.4.45 (Unary Transformation)** *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon and let $f$ be a unary y-function and $\Delta$ 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 $exp_e(i)$ (Def. 2.2.33) is the first non-linear transformation we consider here.

**Definition 2.4.46 (Exponentiation)** *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon, $e$ a non-negative number (the exponent) and $\Delta$ the threshold.*

$I.Exp(e) \overset{\text{def}}{=} I.UnaryTransformation(\lambda(y) y^e, \Delta)$.
$\lambda(y) y^e$ *is not linear.* ■

### Complement Operator

Another point-based transformation is the complement operator.

**Definition 2.4.47 (Complement)** *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon, $n$ a negation function (Def. 2.2.20). and $\Delta$ the threshold.*

$I.Complement(n, \Delta) \overset{\text{def}}{=} I.UnaryTransformation(n, \Delta)$. ■

## 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 2.4.48 (Intersection Points)** *Let $I_1 = (p_0, \dots, p_{n+1})$ and $I_2 = (q_0, \dots, q_{m+1})$ 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 $((p_0, q_0), \dots)$.*
*Let $IntP \overset{\text{def}}{=} ()$.*
*Let $i \overset{\text{def}}{=} 0$ and $j \overset{\text{def}}{=} 0$*
*Let $x \overset{\text{def}}{=} p_0.x$ ($x$ is the position of the sweep line).*

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

     $if(j < m$ 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 parameterized with two line segments instead of one.

**Definition 2.4.49 (Binary Transformation)** *Let $I_1 = (p_0, \dots, p_{n_1})$ and $I_2 = (q_0, \dots, q_{n_2})$ be envelope polygons. Let $f$ be a binary y-function and $\Delta$ a threshold value.*
*We describe the method $I_1.BinaryTransformation(I_2, f, \Delta)$ procedurally:*

```
    Let $I \overset{\text{def}}{=} ((p_0, q_0), \dots, (p_n, q_n)) = I_1.IntersectionPoints(I_2)$
    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 $newI \stackrel{\text{def}}{=} (p_0.x, f(p_0.y, q_0.y))$;
$For_{i=1}^{n}$ $newI.Interpolate(p_i.x, \lambda(x) f(p_{i-1}.y.LineY((p_i.y, x), q_{i-1}.x.LineY((q_i.x, x)), \Delta)$;
return $newI$;
```
$\blacksquare$

## Interval–Interval Relations

Most formulae for interval–interval relations contain integrals over products of membership functions. The corresponding algorithms are listed in this section. All other algorithms for interval–interval relations have already been explained or they are trivial.

**Definition 2.4.50 (Integration over Multiplied Intervals)** *Let $I = (p_0, \ldots, p_n)$ and $J = (q_0, \ldots, q_m)$ be envelope polygons. The integral $I.Integrate(J, a) \stackrel{\text{def}}{=} \int_{-\infty}^{+\infty} I(x-a)J(x)\ dx$ 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 $j < m)$
        $Int = Int + p'_i.Integrate(p'_{i+1}, q_j, q_{j+1}, x, \min(p_{i+1}.x + a, q_{j+1}.x))$;        $//Def.2.4.12$
        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 $Int$;
```

*The next method calls $Integrate$ and normalizes the result.*

$$I.IntegrateAsymmetric(J) \stackrel{\text{def}}{=} roundY\left(\frac{2 \cdot I.Integrate(J, 0)}{I.Size2()}\right)$$

$\blacksquare$

**Definition 2.4.51 (Symmetric Integration)** *Let $I$ and $J$ be two envelope polygons. The function $I.IntegrateSymmetric(J, simple)$ computes $\int_{-\infty}^{+\infty} I(x) \cdot J(x)\ dx/N$*

*where* $N = \begin{cases} \min(|I|, |J|) & \text{if } simple = true \\ \max_a(\int_{-\infty}^{+\infty} I(x-a) \cdot J(x)\ dx) & \text{otherwise} \end{cases}$

*$I.IntegrateSymmetric(J, simple)$*

$$\stackrel{\text{def}}{=} \begin{cases} undefined & \textit{if } I.isInfinite() \textit{ or } J.isInfinite() \\ roundY(\dfrac{2 \cdot I.Integrate(J,0)}{\min(I.Size2(), J.Size2)}) & \textit{if } simple = true \\ roundY(\dfrac{\top \cdot I.Integrate(J,0)}{maximizeOverlap(I,J)} & \textit{otherwise (Def. 2.2.70)} \end{cases}$$

■

## 2.5 The FuTIRe Interface

The FuTIRe interface consists of the classes Point, Interval, YFunction, and lots of subclasses of the class Operation. For each of these 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 (for example long long 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. FuTIRe 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.

### 2.5.1 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 $p \to q \to r$ is a left turn or colinear. (Def. 2.4.4)

`bool p.leftturnProper(Point q, Point r)`
true if $p \to q \to r$ is a proper left turn.

`bool p.rightturn(Point q, Point r)`
true if $p \to q \to r$ is a right turn or colinear.

`bool p.rightturnProper(Point q, Point r)`
true if $p \to q \to r$ is a proper right turn.

`bool p.colinear(Point q, Point r)`
true if $p \to q \to r$ is colinear (Def. 2.4.3)

`bool p.colinear(Point q, Point r, Point s)`
true if the line segment $(p, q)$ is colinear with the line segment $(r, s)$.

`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 $(p, q)$ intersects the line $(r, s)$.

`bool p.intersectsProper(Point q, Point r, Point d)`
: true if the line $(p, q)$ intersects the line $(r, s)$, but does not only touch it. (Def. 2.4.5)

**Computations**

`CX p.intersection(Point q, Point r, Point s)`
: computes the intersection point for the line segments $(p, q)$ and $(r, s)$. An error is thrown if the line segments do not intersect! (Def. 2.4.6)

`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. 2.4.7)

`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. 2.4.8)

`CX p.Area2(Point q)`
: computes the area below the line segment $(p, q)$. (Def. 2.4.9)

`float p.Area2(Point q, CX x1, CX x2)`
: computes the area below the line segment $(p, q)$ from $x_1$ until $x_2$. It throws an error of the line is vertical and $x1 \neq x2$ (Def. 2.4.9)

`CX p.Area2X(Point q, float a)`
: computes the $x$-coordinate $x$ such that the area below the line segment $(p, q)$ from $p$ until $x$ is just $a$. An error is thrown if there is not enough area below the line segment. (Def. 2.4.10)

`float p1.Integrate(Point p2, Point q1, Point q2, CX x1, CX x2)`
: computes $\int_{x1}^{x2} l_1(x) \cdot l_2(x)\ dx$ 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. 2.4.12)

## 2.5.2 Intervals

The Intervals class manages and manipulates fuzzy temporal intervals (Sec. 2.4.2). The intervals are represented by their envelope polygons (Def. 2.4.15).

**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 $[x_1, y_1\ x_2, y_2\ ...[$.

`void close()`
declares the polygon as finished and removes some redundancies. (Def. 2.4.19)

**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. 2.4.18)

`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. 2.4.18)

`void I.pop_back()`
removes the last point from the polygon. It does nothing on empty polygons. (Def. 2.4.18)

**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.

`bool I.isConvex()`
  returns true if the polygon is convex.

`bool I.isMonotone()`
  returns true if the polygon is monotone. (Def. 2.2.68)

`bool I.isSymmetric()`
  returns true if the polygon is symmetric. (Def. 2.2.68)

`CX I.SymmetryAxis()`
  returns the $x$-coordinate of the symmetry axis and throws an error if $I$ is not symmetric.

`int I.Index(CX cx)`
  returns the index of the rightmost polygon point that is left of $x$, or -1 if there is no such point. (Def. 2.4.20)

`int I.MaxIndex(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. 2.4.20)

`CY I.Min()`
  returns the smallest $y$-value of the polygon. (Def. 2.4.27)

`CY I.Max()`
  returns the hight $I\hat{}$ of the polygon. (Def. 2.4.27)

`CX I.MaxX(bool front)`
  returns the $x$-coordinate of the leftmost / rightmost point with maximal $y$-value. It throws an error if the polygon is empty (Def. 2.4.27)

`float I.Member(CX x)`
  returns the membership value for the $x$-coordinate $x$. (Def. 2.4.22)

`CX I.Size2I(int k, int l)`
  returns 2 * the area below the polygon from vertex k to vertex l. (Def. 2.4.29)

`CX I.Size2()`
  returns 2 * the area below the polygon. (Def. 2.4.29)

`CX I.Size2(CX a, CX b)`
  returns 2* the area below the polygon from $x$-coordinate $a$ to $x$-coordinate $b$. (Def. 2.4.29)

`CX I.CenterPoint(int k, int m)`
  returns the $x$-coordinates of the $k, m$-center point. (Def. 2.4.30)

`int I.nComponents()`
  returns the number of components of the interval. (Def. 2.4.32)

`Interval I.Component(int k)`
  returns the $k^{th}$ component of $I$. It throws an error if $k < 0$. (Def. 2.4.33)

**The Core of the Interval (Def. 2.4.34)**

`CX I.CSize()`
  returns the size of the core.

`vector<<int,int>> I.CList()`
: returns the list of indices of the core boundaries.

`Interval I.CCrisp()`
: returns the core as crisp interval.

`CX I.CFirst()`
: returns the $x$-coordinate of the leftmost point of the core.

`CX I.CLast()`
: returns the $x$-coordinate of the rightmost point of the core.

### The Support of the Interval (Def. 2.4.35)

`CX I.SSize()`
: returns the size of the support.

`vector<<int,int>> I.SList()`
: returns the list of indices of the support boundaries.

`Interval I.SCrisp()`
: returns the support as crisp interval.

`CX I.SFirst()`
: returns the $x$-coordinate of the leftmost point of the support. It throws an error if the polygon is empty.

`CX I.SLast()`
: returns the $x$-coordinate of the rightmost point of the support. It throws an error if the polygon is empty.

### The Kernel of the Interval (Def. 2.4.36)

`CX I.KSize()`
: returns the size of the kernel.

`vector<<int,int>> I.KList()`
: returns the list of indices of the kernel boundaries.

`Interval I.KCrisp()`
: returns the kernel as crisp interval.

`CX I.KFirst()`
: returns the $x$-coordinate of the leftmost point of the kernel. It throws an error if the polygon is empty.

`CX I.KLast()`
: returns the $x$-coordinate of the rightmost point of the kernel. It throws an error if the polygon is empty.

### Hull Calculations

`Interval I.CrispHull()`
: returns the crisp hull of $I$. (Def. 2.4.37)

`Interval I.MonotoneHull()`
: returns the monotone hull of $I$. (Def. 2.4.38)

`Interval I.ConvexHull()`
: returns the convex hull of $I$. (Def. 2.4.39)

### Simple Predicates

`bool I.isSubset(CX t1, CX t2)`
: returns true if [t1,t2[ is a subset of $I$'s support.

`bool I.overlaps(CX t1, CX t2)`
: returns true if [t1,t2[ overlaps with $I$'s support.

`bool I.biggerPartInside(CX t1, CX t2)`
: returns true if the bigger part of [t1,t2[ is inside $I$'s support.

### Basic Unary Transformations (Def. 2.2.33)

`Interval I.Extend(true)`
: returns the rising part of $I$. (Def. 2.4.40)

`Interval I.Extend(false)`
: returns the falling part of $I$. (Def. 2.4.40)

`Interval I.ScaleUp()`
: scales the $y$-values of the interval up to $\top$. (Def. 2.4.40)

`Interval I.ScaleUpD()`
: is the destructive version of ScaleUp.

`Interval I.Shift(CX a)`
: shifts the interval by $a$ units. (Def. 2.4.40)

`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. 2.4.41)

`Interval I.CutI(int i1, int i2)`
: cuts the part of the interval between the points with index $i1$ and $i2$. (Def. 2.4.41)

`Interval I.Times(float a)`
: multiplies the $y$-values of the interval by $a$. (Def. 2.4.42)

`Interval I.Exp(float e)`
: exponentiates the $y$-values of the interval with $e$. (Def. 2.4.46)

`Interval I.Integrate(true)`
: computes $\int_{-\infty}^{x} I(y)dy/|I|$. $I$ may be infinite. (Def. 2.4.44)

`Interval I.Integrate(false)`
: computes $\int_{x}^{+\infty} I(y)dy/|I|$. $I$ may be infinite. (Def. 2.4.44)

`Interval I.Negate()`
: inverts the $y$-values. (Def. 2.2.33)

`Interval I.Invert()`
: inverts the $y$-values of the gaps in the interval. (Def. 2.2.33)

`CY I.IntegrateAsymmetric(Interval J)`
: computes $\int I(x) \cdot J(x)\ dx/|I|$. $I$ and $J$ may be infinite. (Def. 2.4.50)

`CY I.IntegrateSymmetric(Interval J,bool simple)`
: computes $\int(x) \cdot J(x)\ dx/N(I,J)$. It throws an error if both $I$ and $J$ are infinite. (Def. 2.4.51)

**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. 2.2.36)

`Interval I.FuzzifyLinear(bool front, float percent, float offset)`
  linear fuzzification of the front/end part of the interval with relative coordinates. (Def. 2.2.39)

`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. 2.2.37)

`Interval I.FuzzifyGaussian(bool front, float percent)`
  Gaussian fuzzification of the front/end part of the interval with relative coordinates. (Def. 2.2.39)

**General Transformations**

`Interval I.UnaryTransformation(UnaryYFunction f)`
  applies the unary y-function $f$ to $I$. (Def. 2.4.45)

`Interval I.BinaryTransformation(Interval J, BinaryYFunction f)`
  applies the binary y-function $f$ to $I$ and $J$. (Def. 2.4.49)

### 2.5.3 Y-Functions

The unary and binary transformation methods (Def. 2.4.45, 2.4.49) 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 $\lambda$-Complement (Def. 2.2.21) $n_\lambda(y)\overset{\text{def}}{=}\frac{1-y}{1+\lambda y}$ depends on the parameter $\lambda$. This would not be a problem in most functional programming languages. One can define $n(\lambda, x)$ and then get $n_\lambda$ 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.

FuTIRe realizes y-functions with the class hierarchy in Fig. 2.1.

The top class, 'Operator', manages the mapping of function names to the functions (instances of the other classes). Each instance can get a name, for example 'myFavorite-LambdaComplement', and one can retrieve the corresponding instance with the method `Operator::getByName(string name)`. The name is optional. Instances without names are not accessible via getByName.

**Constructors**

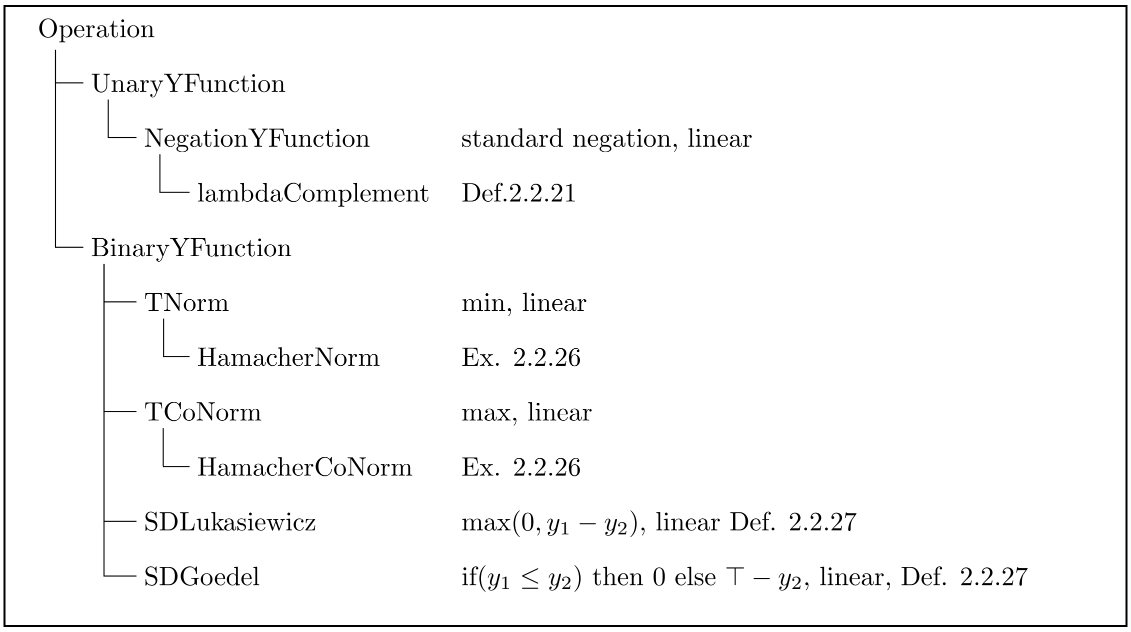

Operation
UnaryYFunction
NegationYFunction standard negation, linear
lambdaComplement Def.2.2.21
BinaryYFunction
TNorm min, linear
HamacherNorm Ex. 2.2.26
TCoNorm max, linear
HamacherCoNorm Ex. 2.2.26
SDLukasiewicz max(0, y1 − y2), linear Def. 2.2.27
SDGoedel if(y1 ≤ y2) then 0 else ⊤ − y2, linear, Def. 2.2.27

Figure 2.1: Class Hierarchy for Y-Functions

`NegationYFunction(string name)`
: constructs the standard negation function $\lambda(y)(1-y)$. (Def. 2.2.21)

`lambdaComplement(float lambda, string name)`
: constructs the lambda complement $\lambda(y)\frac{1-y}{1+\lambda y}$. (Def. 2.2.21)

`TNorm(string name)`
: Constructs the min t-norm.

`HamacherNorm(float gamma, string name)`
: Constructs the Hamacher t-norm $\lambda(x,y)\frac{xy}{\gamma+(1-\gamma)(x+y-xy)}$. (Def. 2.2.26)

`TCoNorm(string name)`
: Constructs the max t-conorm

`HamacherCoNorm(float beta, string name)`
: Constructs the Hamacher t-conorm $\lambda(x,y)\frac{x+y+(\beta-1)xy}{1+\beta xy}$. (Def. 2.2.26)

**Parameter Modification**
The parameters lambda, gamma, beta are public instance variables and can therefore be changed by direct assignment.

### 2.5.4 Interval Operators

The interval operators transform one or two given intervals into a new interval. Some of them are implemented as methods of the Interval class. Since methods are in most programming languages not first-class objects, this has some disadvantages. The main disadvantages are that it is not possible to have methods as parameters of other methods, and to construct new methods at run time by composing existing ones. Interval operators in FuTIRe are therefore represented as instances of corresponding classes. This way it is no problem to have an interval operator which has another interval operator as parameter, and to combine them in various ways. The class 'IntervalOperator' is a subclass of the class 'Operator' which we have met already. The full class hierarchy is presented in Fig. 2.2.

A very specific class is the class UIODefined. Its constructor function allows one to define paramterized compositions of unary interval operators. The syntax of the definitions is presented in Section. 2.5.5 below.

```
Operation
 │
 └──── IntervalOperator
         ├──── UnaryIntervalOperator
         │       ├──── UIOComposeUnary
         │       ├──── UIOComposeBinary
         │       ├──── UIODefined
         │       │     UIObyYFunction
         │       │       └──── UIOComplement
         │       ├──── UIOBasic
         │       │       ├──── UIOExtend,UIOScaleUp,UIOShift,UIOCut,UIOCutI,UIOTimes
         │       │       └──── UIOExp,UIOIntegrate,UIOInvert,UIOComponent,UIOFuzzify,UIOHull
         │       └──── PIRelation
         │               ├──── PIRBefore
         │               │       └──── PIRFuzzyBefore
         │               ├──── PIRAfter
         │               │       └──── PIRFuzzyAfter
         │               ├──── PIRStarts
         │               │       └──── PIRFuzzyStarts
         │               ├──── PIRFinishes
         │               │       └──── PIRFuzzyFinishes
         │               ├──── PIRDuring
         │               │       └──── PIRDuring
         │               ├──── PIRInTheMiddle
         │               │       └──── PIRFuzzyInTheMiddle
         │               └──── PIRInTheGap
         │                       └──── PIRFuzzyInTheGap
         └──── BinaryIntervalOperator
                 ├──── BIObyYFunction
                 │       ├──── BIOUnion
                 │       ├──── BIOIntersection
                 │       └──── BIOSetdifference
                 │               └──── BIOFuzzySetdifference
                 └──── BIOUntil
```

Figure 2.2: Class Hierarchy for Interval Operators

**General Methods for Unary Interval Opertors:**

`operator()(Interval I)`
: If $f$ is an instance of one of the unary interval operator classes and $I$ an interval then $f(I)$ yields the transformed interval.

`apply(CX x, Interval I, float e)`
: If $f$ is an instance of one of the unary interval operator classes below and $I$ and interval then $f.apply(x, I, e)$ yields $f(I)(x)^e$.

**General Methods for Binary Interval Opertors:**

`operator()(Interval I, Interval J)`
: If $f$ is an instance of one of the unary interval operator classes and $I$ and $J$ are intervals then $f(I, J)$ yields the transformed interval.

`apply(CX x, Interval I, Interval J, float e)`
: If $f$ is an instance of one of the binary interval operator classes and $I$ and $J$ are intervals then $f.apply(x, I, J, e)$ yields $f(I, J)(x)^e$.

**Constructors**

All constructors below have an optional 'string name' argument as last argument.

`UIOComposeUnary(UnaryIntervalOperator U1, UnaryIntervalOperator U2)`
: realizes the composition $U1 \circ U2$ of two unary interval operators.

`UIOComposeUnary(vector<(UnaryIntervalOperator> Ops)`
: realizes the composition of all unary interval operators in *Ops*.

`UIOComposeBinary(UnaryIntervalOperator U1, UnaryIntervalOperator U2, BinaryIntervalOperator B)`
: realizes the composition $B(U1(I), U2(I))$ of two unary and one binary interval operator.

`UIODefined(string definition)`
: parses the definition and creates a parameterized unary interval operator. The definition is parsed by the parser generated from the files Parser.y and Scanner.l, which contain the grammar in flex/bison notation. (Def. 2.5.1)

`UIObyYFunction(UnaryYFunction F)`
: generates an operator which just calls $I.UnaryTransformation(F)$. (Def. 2.4.45)

`BIObyYFunction(BinaryYFunction F)`
: generates an operator which just calls $I.UnaryTransformation(J, F)$. (Def. 2.4.49)

`UIOComplement()`
: Generates the standard Complement Operator $\lambda(y)(1 - x)$.

`UIOComplement(float lambda)`
: Generates the $\lambda$-Complement Operator $\lambda(y)\frac{1-y}{1+\lambda y}$. (Def. 2.2.21)

`UIOExtend(bool rising)`
: generates an operator which returns the rising part of an interval, if $rising = true$, otherwise it generates the falling part (see $I.Extend(rising)$). (Def. 2.4.40)

`UIOScaleUp()`
: generates an operator which scales the $y$-values of the interval up to $\top$. (Def. 2.4.40)

`UIOShift(CX a)`
: generates an operator which shifts the interval by $a$ units (see $I.Shift(a)$.) (Def. 2.4.40)

`UIOCut(CX x1, CX x2)`
: generates an operator which cuts the part of the interval between $x1$ and $x2$ (see $I.Cut(x1, x2)$). (Def. 2.4.41)

`UIOCut(CX x1, bool positive)`
: generates a cut operator which, if positive = true, works like UIOCut(x1,$+\infty$), otherwise works like UIOCut($-\infty$,x1) (Def. 2.4.41)

`UIOCutI(int i1, int i2)`
: generates an operator which cuts the part of the interval between the points with index $i1$ and $i2$ (see $I.CutI(i1, i2)$. (Def. 2.4.41)

`UIOTimes(float a)`
: generates an operator which multiplies the $y$-values of the interval by $a$ (see $I.Times(a)$). (Def. 2.4.42)

`UIOExp(float e)`
: generates an operator which exponentiates the $y$-values of the interval with $e$ (see $I.Exp(e)$). (Def. 2.4.46)

`UIOIntegrate(bool rising)`
: generates an operator which integrates the membership function, from $-\infty$, if $rising = true$, otherwise from $+\infty$ (see $I.Integrate(front)$). (Def. 2.4.44)

`UIOInvert()`
: generates an operator which inverts the $y$-values (see $I.Invert()$) (Def. 2.2.33)

`UIOComponent(int k)`
: generates an operator which extracts the $k^{th}$ component from the interval (see $I.Component(k)$.) (Def. 2.4.33)

`UIOFuzzify(true, bool front, CX x1, CX x2, CX offset)`
: generates a linear fuzzify operator with absolute coordinates (Def. 2.2.36)

`UIOFuzzify(true, bool front, float percent, float offset)`
: generates a linear fuzzify operator with relative coordinates (Def. 2.2.39)

`UIOFuzzify(false, bool front, CX x1, CX x2, CX offset)`
: generates a Gaussian fuzzify operator with absolute coordinates (Def. 2.2.37)

`UIOFuzzify(false, bool front, float percent)`
: generates a Gaussian fuzzify operator with relative coordinates (Def. 2.2.39)

The name of the first parameter in the UIOFuzzify-method is 'linear'.

`UIOHull(version)`
: generates an operator which generates the corresponding hull. The following versions are avaiable: crisp (Def. 2.4.37), monotone (Def. 2.4.38), convex (Def. 2.4.39), core (Def. 2.4.34), support (Def. 2.4.35) and kernel (Def. 2.4.36).

`BIOUnion()`
: generates an operator which computes the union of two fuzzy intervals using the max-norm (Def. 2.2.24)

`BIOUnion(TCoNorm S)`
: generates an operator which computes the union of two fuzzy intervals using the t-conorm $S$. (Def. 2.2.24)

`BIOIntersection()`
: generates an operator which computes the intersection of two fuzzy intervals using the min-norm (Def. 2.2.24)

`BIOIntersection(TNorm T)`
: generates an operator which computes the union of two fuzzy intervals using the t-norm $T$. (Def. 2.2.24)

`BIOSetdifference(SDVersion Version)`
: generates an operator which computes the Kleene, Lukasiewicz, Goedel set difference of two fuzzy sets. SDVersion is an enumeration type with elements (Kleene, Lukasiewicz, Goedel) (Def. 2.2.29)

`BIOFuzzySetdifference(BIOIntersection T, BIOComplement C)`
: generates an operator which computes the generalized Kleen set difference of two fuzzy sets: $I \setminus J \overset{\text{def}}{=} T(I, C(J))$. (Def. 2.2.27)

`BIOUntil(bool begin, bool end)`
: generates an *Until* operator with $E^+ = UIOExtend(begin)$, $E^- = UIOExtend(end)$, standard intersection and standard complement. (Def. 2.2.53)

`BIOUntil(bool begin, bool end, UnaryIntervalOperator* E1,`
`UnaryIntervalOperator* E2,`
`BIOIntersection* Ints, UIOComplement* Cmpl)`
: generates a general *Until* operator (Def. 2.2.53)

**Constructors for the Point–Interval Relation Operators**

The point–interval relation operators come in two versions, for example PIRBefore and PIRFuzzyBefore. The first version works like the corresponding standard crisp relations. As boundaries of the intervals one can choose either the support, the core or the kernel. PIRFuzzyBefore is the operator version $N(E^+(I))$.

CrVersion is an enumeration type with values (Support,Core,Kernel)

`PIRBefore(CrVersion Version)`
: generates a crisp *before* operator $x$ *before* $I$ iff $x < I^{fV}$ where $V$ is determined by the $Version$ parameter. (Def. 2.2.42)

`PIRFuzzyBefore()`
: generates a *before* operator $N(E^+(I))$ with $E^+ = UIOExtend(true)$ and standard complement $N$. (Def. 2.2.43)

`PIRFuzzyBefore(UnaryIntervalOperator Ep, Complement N)`
: generates a *before* operator $N(Ep(I))$. (Def. 2.2.43)

`PIRAfter(CrVersion Version)`
: generates a crisp *after* operator $x$ *after* $I$ iff $x > I^{lV}$ where $V$ is determined by the $Version$ parameter. (Def. 2.2.42)

`PIRFuzzyAfter()`
: generates an *after* operator $N(E^-(I))$ with $E^- = UIOExtend(false)$ and standard complement $N$. (Def. 2.2.43)

`PIRFuzzyAfter(UnaryIntervalOperator Em, Complement N)`
: generates an *after* operator $N(Em(I))$. (Def. 2.2.43)

`PIRStarts(CrVersion Version)`
: generates a *starts* operator $x$ *starts* $I$ iff $x = I^{fV}$ where $V$ is determined by the $Version$ parameter. (Def. 2.2.42)

`PIRFuzzyStarts()`

generates a *starts* operator $scaleup(Ep(I)\ T\ B(I))$ where $Ep = UIOExtend(true)$, $T = BIOIntersection()$ and $B = PIRFuzzyBefore()$. (Def. 2.2.43)

`PIRFuzzyStarts(UnaryIntervalOperator Ep, PIRBefore B, BIOIntersection T)`

generates a *starts* operator $scaleup(Ep(I)\ T\ B(I))$. (Def. 2.2.43)

`PIRFinishes(CrVersion Version)`

generates a *finishes* operator $x$ *finishes* $I$ iff $x = I^{lV}$ where $V$ is determined by the $Version$ parameter. (Def. 2.2.42)

`PIRFuzzyFinishes()`

generates a *finishes* operator $scaleup(Em(I)\ T\ A(I))$ where $Em = UIOExtend(false)$, $T = BIOIntersection()$ and $A = PIRAfter()$. (Def. 2.2.43)

`PIRFuzzyFinishes(UnaryIntervalOperator Em, UIOAfter A, Intersection T)`

generates a *finishes* operator $scaleup(Ep(I)\ T\ A(I))$. (Def. 2.2.43)

`PIRDuring()`

generates the identity operator as the standard *during* operator.

`PIRDuring(int n, int m)`

generates a $during_{n,m}$ operator $x$ *during* $I$ iff $x \in I.cut_{I^{n,m},I^{n+1,m}}$.

`PIRFuzzyDuring(UnaryTransformation O, BIOUnion U)`

generates the *during* operator $U_{l \in CMP(I)} O(I)$. (Def. 2.2.43)

`PIRFuzzyDuring(PIRDuring D, int n, int m)`

generates the *during* operator $D(cut_{I^{n,m},I^{n+1,m}})$. (Def. 2.2.43)

`PIRInTheMiddle(int n, int m)`

generates an *in_the_middle* operator $x in\_the\_middle_{n,m}(I)$ iff $x = I^{2n+1,2m}$

`PIRFuzzyTheMiddle(PIRDuring D, int n, int m, int k)`

generates the *in_the_middle* operator $D(cut_{i^{2^k(2n+1)-1,2^k 2m},i^{2^k(2n+1)+1,2^k 2m}}(I))$. (Def. 2.2.43)

`PIRInTheGap(CrVersion Version)`

generates an *in_the_gap* operator $x in\_the\_gap(I)$ iff $x \in V(Invert(I))$ where $V$ is determined by the $Version$ parameter.

`PIRInTheGap(CrVersion Version, int k)`

generates an $in\_the\_k^{th} gap$ operator $x in\_the\_k^{th} gap(I)$ iff $x \in V(Component(Invert(I), k))$ where $V$ is determined by the $Version$ parameter.

`PIRFuzzyInTheGap(PIRDuring D)`

generates an *in_the_gap* operator $D(invert(I))$ (Def. 2.2.43)

`PIRFuzzyInTheGap(PIRDuring D, int k)`

generates an $in\_the\_k^{th} gap$ operator $D(Component(invert(I), k))$ (Def. 2.2.43)

### 2.5.5 Specification of Unary Interval Operators

The class UIOComposeUnary allows one to create the composition $U_1 \circ \ldots \circ U_n$ of unary interval operators. This is a very primitive kind of composition because the implicit parameters of the component operators are fixed then.

A much more flexible way of composition is provided by the class UIODefined. Its constructor function accepts a specification of a composition of unary interval operators with free variables. When the variables are bound to corresponding values then the specified abstract composition is instantiated to a concrete unary interval operator.

As an example, suppose we want a flexible operator which fuzzifies an interval at both sides. A suitable definition would be:

```
UIODefined("fuzzify2sides(Keyword version, Float percent) =
   fuzzify(version,front,percent) * fuzzify(version,back,percent)")
```

This defines a function with two parameters. The first parameter `version` can be `linear` or `gaussian`. The second parameter `percent` represents a floating point number which determines the increase of the membership function. The constructed instance of UIODefined understands an `instantiate` method which binds the two parameters `version` and `percent` to concrete values. This turns the function into a valid unary interval operator which can be applied to a fuzzy interval.

**Definition 2.5.1 (Syntax of Defined Operators)** *The definitions which are accepted by the UIODefined constructor have the following structure:*

$\texttt{name(type}_1\ \texttt{variable}_1\texttt{, ...type}_n\ \texttt{variable}_n\texttt{) = term}_1\ \texttt{* ...* term}_k$

*The* $\texttt{variable}_i$ *are variable names (strings). The* $\texttt{type}_i$ *are names of basic data types. The following names can be used here:* `integer, float, bool, time, keyword`*. Integer and Float and Bool stand for the usual data types.* `time` *stands for x-coordinates. The concrete data type, e.g. long long int, or multiple precision integer is determined by the compiler.* `keyword` *represents keywords (strings) like 'linear' or 'forward' etc.*

*The* $\texttt{term}_i$ *have the form* `name` *or* $\texttt{name(parameter}_1\texttt{,..., parameter}_j\texttt{)}$*.* `name` *is the name of a concrete unary interval operator. The* $\texttt{parameter}_i$ *are either variable names, a the string representation of concrete data of the corresponding data types, or again terms term*$_1$ ** ... * term*$_l$ *which determine a unary interval operator.*

*The following terms are allowed:*

- `complement` *generates the standard complement operator* `UIOComplement()`*.*
- `complement(lambda)` *generates the lambda complement operator* `UIOComplement(lambda)` *(Def. 2.2.21). The type of* `lambda` *must be float.*
- `extend(rising)` *generates the* `UIOExtend(rising)` *operator.* `rising` *is a keyword which is either 'rising' or 'falling' (Def. 2.4.40).*
- `scaleup` *generates the* `UIOScaleUp()` *operator (Def. 2.4.40).*
- `shift(a)` *generates the* `UIOShift(CX a)` *operator which shifts the interval by the amount* `a`*.* `a` *is of type* `time` *(Def. 2.4.40).*
- `invert` *generates the* `UIOInvert()` *operator which inverts the y-values (Def. 2.2.33).*
- `cut(x1,x2)` *generates the* `UIOCut(x1,x2)` *operator which cut the piece between* `x1` *and* `x2` *out of the interval.* `x1` *and* `x2` *are of type* `Time` *(Def. 2.4.40).*
- `cut(x,direction)` *generates the* `UIOCut(x,positive)` *operator.* `x` *is of type* `time`*, and* `direction` *is one of the keywords 'forward' or 'backward'.* `cut(x,forward)` = $\texttt{cut(x,}+\infty\texttt{)}$ *and* `cut(x,backward)` = $\texttt{cut(-}\infty\texttt{,x)}$ *(Def. 2.4.41).*

- `times(a)` *generates the* `UIOTimes(a)` *operator which multiplies the fuzzy values of the interval by* `a`. `a` *is of type* `float` *(Def. 2.4.46).*
- `integrate(rising)` *generates the* `UIOIntegrate(rising)` *operator which integrates the membership function.* `rising` *is a keyword which is either 'rising' or 'falling' (Def. 2.4.44).*
- `component(k)` *generates the* `UIOComponent(k)` *operator which extracts the* $k^{th}$ *component from the interval. The type of* `k` *is integer (Def. 2.4.33).*
- `hull(version)` *generates the* `UIOHull(version)` *operator.* `version` *is one of the keywords 'crisp' (Def. 2.4.37), 'monotone' (Def. 2.4.38), 'convex' (Def. 2.4.39), 'core' (Def. 2.4.34), 'support' (Def. 2.4.35) or 'kernel' (Def. 2.4.36). The operator computes the corresponding hull.*
- `fuzzify(version,side,percent,offset)` *generates the (relative) UIOFuzzify operator.* `version` *is one of the keywords 'linear' or 'gaussian'.* `side` *is one of the keywords 'front' or 'back'. It determines the side of the interval to be fuzzified.* `percent` *is a float value which determines the fraction of the interval (in %) to be fuzzified.* `offset` *is also a float value and determines a shift (in %) of the fuzzified part of the interval (Def. 2.2.39).*
- `fuzzify(version,side,x1,x2,offset)` *generates the (absolute) UIOFuzzify operator.* `version` *and* `side` *are the same as above.* `x1` *and* `x2` *are of type time and determine the part of the interval to be fuzzified.* `offset` *determines the shift of the fuzzified part of the interval in absolute coordinates (Def. 2.2.36, 2.2.37).*
- `before(version)` *generates the crisp point-interval* before *relation PIRBefore(version).* `version` *is one of the keywords 'support', 'core' or 'kernel' and determines the crisp hull of the interval for the crisp before relation (Def. 2.2.42).*
- `before` *is an abbreviation for* `before(support)`.
- `fuzzyBefore(operators,complement)` *generates a fuzzy* before *operator PIRFuzzyBefore (operators, complement) (Def. 2.2.43).* `operators = term`$_1$ `* ...` *is the specification of a unary interval operator. Typically* `operators = extend`. `complement` *determines the complement operator. It is either just* `complement`*, for the standard complement operator, or* `complement(lambda)` *for the lambda complement operator.*
- `fuzzyBefore` *is a shortcut for* `fuzzyBefore(extend(rising),complement)`.
- `after(version)` *(Def. 2.2.42) corresponds to* `before(version)`.
- `after` *is an abbreviation for* `after(support)`.
- `fuzzyAfter(operators,complement)` *corresponds to* `fuzzyBefore(operators,complement)`.
- `fuzzyAfter` *is a shortcut for* `fuzzyBefore(extend(falling),complement)`.
- `starts(version)` *generates a crisp point-interval* starts *operator.* `version` *is one of the keywords 'support', 'core' or 'kernel' and determines the crisp hull of the interval for the crisp before relation (Def. 2.2.42).*

- `starts` *is a shortcut for* `starts(support)`.
- `fuzzyStarts(Extend,Before,gamma)` *generates a fuzzy* before *operator.* `Extend` *can be the specification of an extend operator, e.g.* `extend(rising)`, *or the specification of an arbitrary unary interval operator.* `Before` *is the specification of a crisp or fuzzy* before *operator like* `before(support)`. `gamma` *is the parameter of the Hamacher intersection. The generated operator is* $scaleup(Extend(I)\ T_{gamma}\ Before(I))$ *(Def. 2.2.43).*
- `fuzzyStarts` *generates a standard fuzzy* starts *operator where* $Extend = UIOExtend(true)$, $Before = PIRFuzzyBefore()$ *and standard intersection (Def. 2.2.43).*
- `finishes(version)` *corresponds to* `starts(version)`.
- `finishes` *is a shortcut for* `finishes(support)`.
- `fuzzyFinishes(Extend, After, gamma)` *corresponds to* `fuzzyStarts(Extend, Before, gamma)` *(Def. 2.2.43).*
- `fuzzyFinishes` *corresponds to* `fuzzyStarts`.
- `during` *generates the identity operator as the standard* during *operator.*
- `during(n,m)` *generates a* $\text{during}_{n,m}$ *operator* $x$ during $I$ *iff* $x \in I.cut_{I^{n,m},I^{n+1,m}}$. `n` *and* `m` *are of type integer.*
- `fuzzyDuring(operator,beta)` *generates a fuzzy* during *operator.* `operator` *is the specification of a unary interval operator and* `beta` *(of type float) is the parameter of the Hamacher union. The* `operator` *is applied to the components of the interval and the results are joined together with the union operator (Def. 2.2.43).*
- `fuzzynmDuring(operator,n,m)` *generates an operator which applies* `operator` *to* $cut_{I^{n,m},I^{n+1,m}}$) *(Def. 2.2.43).*
- `inTheMiddle(n,m)` *generates a* `PIRInTheMiddle(n, m)` *operator which is 1 for an interval* $I$ *and a point* $x$ *if* $x = I^{2n+1,2m}$, *i.e.* $x$ *is in the middle of the* $n^{th}$ $m^{th}$.
- `fuzzyInTheMiddle(n,m,k,during)` *generates a fuzzyInTheMiddle operator* $during(cut_{i^{2^k(2n+1)-1,2^k 2m},i^{2^k(2n+1)+1,2^k 2m}}(I))$ ($\overset{\text{def}}{=}$. *2.2.43).*
- `fuzzyInTheMiddle(n,m,k)` *is like the previous operator with the standard during operator as default.*
- `inTheGap(Version)` *generates an 'in the gap' operator* $V(Invert(I))$ *where* $V$ *is determined by the* `Version` *parameter.* `Version` *is one of the keywords 'support', 'core' or 'kernel'.*
- `inTheGap(Version,k)` *generates an 'in the gap' operator* $V(Component(Invert(I),k))$ *where* $V$ *is determined by the* `Version` *parameter.* `Version` *is one of the keywords 'support', 'core' or 'kernel'.*
- `fuzzynTheGap(during)` *generates a 'fuzzy in the gap' operator* $during(Invert(I))$

- `fuzzynTheGap(during,k)` *generates a 'fuzzy in the gap' operator* $during(component(invert(I), k))$.
- ⟨`name`⟩`(parameter`$_1$`,...,parameter`$_n$`)` *invokes a previously defined UIODefined instance with the given name and the given parameters.*

■

The UIODefined class provides a method `instantiate(vector<WCvalue>)` for instantiating the free variables of the defined unary interval operator. `WCvalue` is a union type an consists of the types int, float, bool and void* (pointer). The pointer type void* can be used to point to string objects (for the keywords) or to CXWrapper objects for $x$-coordinates. (The CXWrapper class is a wrapper class for CX values. This technical trick makes it possible to exchange the data structures for CX values, and to use for example long long int or alternatively multiple precision integers.) The vector of `WCvalue` must contain the actual values of the free variables in the same order as in the definition. After the `instantiate` method is applied to a defined interval operator, one can use it as a unary interval operator in the same way as all the other operators.

### 2.5.6 Interval–Interval Relations

The interval–interval relation operators are also implemented as subclasses of the class Operation. They come in three versions. For example, $IIRBefore(Version)$ generates a crisp relation where either the support, the core or the kernel of the two intervals is compared. The subclass $IIRFuzzyBefore(\ldots)$ generates the operator version of the *before*-relation (Def. 2.2.56). Finally the class $IIRFuzzyNMBefore(\ldots)$ generates the Nagypál and Motik version of the *before* relation.

**Methods:**

`operator()(Interval I, Interval J, float e)`
: If, for example, $r$ is an instance of one of the IIRelation classes and $I$ and $J$ are intervals then $r(I,J)^e$ yields the resulting fuzzy value.

**Constructors for the Interval–Interval Relation Operators**
All constructors have an optional 'string name' parameter as last argument.

`IIRBefore(CrVersion Version)`
: generates a *before* operator $I$ *before* $J$ iff $I^{lV} < J^{fV}$ where $V$ is determined by the $Version$-parameter. (Def. 2.2.55)

`IIRFuzzyBefore(PIRBefore B)`
: generates a *before* operator which is essentially $\int I(x) \cdot B(J)(x)\ dx/|i|$ (Def. 2.2.56)

`IIRFuzzyBefore()`
: generates a *before* operator with $B = newPIRFuzzyBefore()$

`IIRNMBefore()`
: generates the Nagypál and Motik *before* operator.

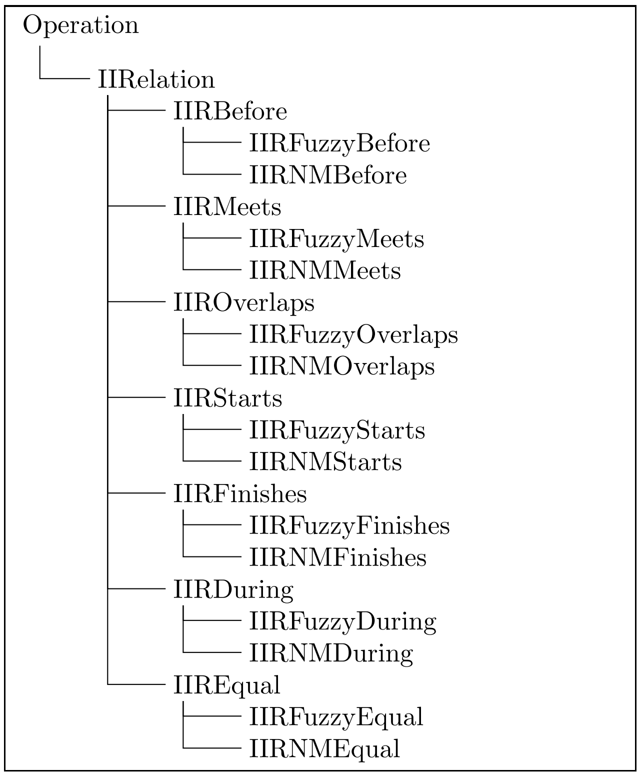


Figure 2.3: Class Hierarchy for Interval–Interval Relations

`IIRMeets(CrVersion Version)`
: generates a *meets* operator $I$ *meets* $J$ iff $I^{lV} = J^{fV}$ where $V$ is determined by the $Version$-parameter. (Def. 2.2.55)

`IIRFuzzyMeets(PIRFinishes F, PIRStarts S, bool simple)`
: generates a *meets* operator which is essentially $\int F(I)(x) \cdot S(I)(x)\ dx / N(F(I), S(J))$ where $N(I,J) = min(|I|, |J|)$ if $simple = true$ and $N = \max_a \int I(x-a) \cdot J(x)\ dx$ otherwise (Def. 2.2.56)

`IIRFuzzyMeets(bool simple)`
: generates the *meets* operator with $F = newPIRFinishes()$ and $S = newPIRFuzzyStarts()$

`IIRNMMeets()`
: generates the Nagypál and Motik *meets* operator.

`IIROverlaps(CrVersion Version)`
: generates a *overlaps* operator $I$ *overlaps* $J$ iff some part $I_1 \subseteq V(I)$ is before $J^{fV}$ and for the rest $I_2 = V(I) \setminus I_1$: $I_2 \subseteq V(J)$ holds. $V$ is determined by the $Version$-parameter. (Def. 2.2.55)

`IIRFuzzyOverlaps(UnaryIntervalOperator* Ep, IIRDuring D, bool simple)` generates an *overlaps* operator which is essitially $(1 - D(I, E^+(J))) * D(I, J)$. If simple = false then this result is nomalized with $\max_a(1 - D(I.shiftD(a), E^+(J))) * D(I.shiftD(a), J)$. (Def. 2.2.56)

`IIRFuzzyOverlaps()`
: generates an *overlaps* operator with $Ep = newUIOExtend(true)$ and $D = newIIRFuzzyDuring()$

`IIRNMOverlaps()`
: generates the Nagypál and Motik *overlaps* operator.

`IIRStarts(CrVersion Version)`
: generates a *starts* operator $I$ *starts* $J$ iff $I^{fV} = J^{fV}$ and $V(I) \subseteq V(J)$. $V$ is determined by the $Version$-parameter. (Def. 2.2.55)

`IIRFuzzyStarts(PIRStarts S1, PIRStarts S2, PIRDuring D, bool simple)`
: generates a *starts* operator which is essentially $\int S_1(I)(x) \cdot S_2(I)(x)\ dx / N(S_1(I), S_2(I)) \cdot D(I, J)$ where $N(I, J) \overset{\text{def}}{=} min(|I|, |J|)$ if $simple = true$ and $N(I, J) \overset{\text{def}}{=} \max_a \int I(x - a) \cdot J(x)\ dx$ otherwise. (Def. 2.2.56)

`IIRFuzzyStarts(bool simple)`
: generates a *starts* operator with $S1 = S2 = newPIRFuzzyStarts()$ and $D = newIIRFuzzyDuring()$

`IIRNMStarts()`
: generates the Nagypál and Motik *starts* operator.

`IIRFinishes(CrVersion Version)`
: generates a *finishes* operator $I$ *finishes* $J$ iff $I^{lV} = J^{lV}$ and $V(I) \subseteq V(J)$. $V$ is determined by the $Version$-parameter. (Def. 2.2.55)

`IIRFuzzyFinishes(PIRFinishes F1, PIRFinishes F2, PIRDuring D, bool simple)`
: generates an *finishes* operator which is essentially $\int F_1(I)(x) \cdot F_2(I)(x)\ dx / N(F_1(I), F_2(I)) \cdot D(I, J)$ where $N(I, J) \overset{\text{def}}{=} min(|I|, |J|)$ if $simple = true$ and $N(I, J) \overset{\text{def}}{=} \max_a \int I(x - a) \cdot J(x)\ dx$ otherwise. (Def. 2.2.56)

`IIRFuzzyFinishes(bool simple)`
: generates a *finishes* operator with $F1 = F2 = newPIRFuzzyFinishes()$ and $D = newIIRFuzzyDuring()$

`IIRNMFinishes()`
: generates the Nagypál and Motik *finishes* operator.

`IIRDuring(CrVersion Version)`
: generates a *during* operator $I$ *during* $J$ iff $V(I) \subseteq V(J)$. $V$ is determined by the $Version$-parameter. (Def. 2.2.55)

`IIRFuzzyDuring(PIRDuring D)`
: generates a *during* operator which is essentially $\int I(x) \cdot D(J)(x)\ dx|I|$ (Def. 2.2.56)

`IIRFuzzyDuring()`
: generates an *during* operator with $D = newPIRFuzzyDuring()$.

`IIRNMDuring()`
: generates the Nagypál and Motik *during* operator.

`IIREquals(CrVersion Version)`
: generates a *equals* operator $I$ *equals* $J$ iff $V(I) = V(J)$. $V$ is determined by the $Version$-parameter. (Def. 2.2.55)

`IIRFuzzyEquals(IIRDuring D)`
: generates an *equals* operator: $D(i,j) \cdot D(j,i)$ (Def. 2.2.56)

`IIRFuzzyEquals()`
: generates an *equals* operator with $D = newIIRFuzzyDuring()$.

`IIRNMEquals()`
: generates the Nagypál and Motik a *equals* operator.

## 2.6 Summary

This report is a detailed description of the FuTIRe-library. So far this library is a C++-package for representing and manipulating fuzzy time intervals and for generating customized point–interval and interval–interval relations for fuzzy intervals (a Java version is planned). The mathematical background, the concrete datastructures and algorithms, and the interface to the library are described.

There are quite different schemes for fuzzy point–interval and interval–interval relations. Besides a crisp approximation for these relations I proposed and implemented in FuTIRe an operator-based scheme which is very flexible and easy to adapt to the needs of concrete applications. The architecture of FuTIRe is open to add other schemes, and to use them together with the crisp and operator-based schemes.

# Chapter 3

# Modelling Periodic Temporal Notions by Labelled Partitionings of the Real Numbers – The PartLib Library

**Hans Jürgen Ohlbach**

## Abstract

The key notion for modelling calendar systems and many other periodic temporal notion is the mathematical concept of *a partitioning of the real numbers*. A partitioning of $\mathbb{R}$ splits the time axis into a sequence of intervals. Basic time units like seconds, minutes, hours, days, weeks, months, years etc. can all be represented by partitionings of $\mathbb{R}$ with finite partitions. Besides the basic time units in calendar systems, there are a lot of other temporal notions which can be modelled as partitions: the seasons, the ecclesiastical calendars, financial years, semesters at universities, the sequence of sunrises and sunsets, the sequence of the tides, the sequence of school holidays etc. In this chapter a formalization of periodic temporal notions by means of *labelled partitionings* of $\mathbb{R}$ is presented. The formalism is implemented as the C++ library *PartLib* (Partitioning Library). The interface to PartLib is presented in the appendix.

## 3.1 Motivation and 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 [16] who analyzed existing calendar systems and came up with algorithms for converting time 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 [10]. 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 [8, 35, 29, 48, 30, 34, 18, 9, 19, 11, 25]. Since time granularities are the most important objects in this area, we introduce them already at this early place in the chapter. 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 [19]. 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 axiomatizing 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 formalization of time loops is a prominent example for this approach [13, 14, 15].

This chapter presents an alternative to the granularities approach. We represent periodic temporal notions as partitionings of the real numbers, which is the simplest form of granularities. To compensate for this, we introduce names (labels) for the partitions. The labels carry information about the meaning of the partitions. As we shall see, this separation of structure and meaning has a number of algorithmic advantageous. A built-in label is 'gap'. It can be used to denote a partition which logically does not belong to a given partitioning. For example, the time between two subsequent school holidays in a school holiday partitioning can be labelled 'gap'. The label 'gap' allows one to simulate the granules of time granularities and nevertheless keep the algorithmic advantages.

The work on partitionings for modelling periodic temporal notions is part of the WebCal-project. WebCal [12] is a system for understanding, representing and manipulating complex temporal notions from everyday life. It can represent and manipulate fuzzy time intervals [40] to deal with fuzzy notions like 'early today' or 'in the late 20s'. It can deal with different calendar systems, even with historical sequences of calendar systems. Other components are a specification language for specifying application specific temporal notions like 'my weekend' [37, 38]. A prototype of the WebCal system is currently being tested.

The guidelines for the particular approach presented in this chapter, and realized in the PartLib library were:

**1. The reality should be taken serious:**
This means that all phenomena in real calendar systems and realistic periodic temporal notions should be taken into account. The consequences of this can be illustrated with the following definition of *month* taken from [35]: The authors defined `day` first, then
`pseudomonth` $= \mathit{Alter}^{12}_{11,-1}(\mathit{day}, \mathit{Alter}^{12}_{9,-1}(\mathit{day}, \mathit{Alter}^{12}_{6,-1}(\mathit{day}, \mathit{Alter}^{12}_{4,-1}($
$\mathit{day}, \mathit{Alter}^{12}_{2,-3}(\mathit{day}, \mathit{Group}_{31}(\mathit{day}))))))$
and finally
`month` $= \mathit{Alter}^{12\cdot400}_{2+12\cdot399,1}(\mathit{day}, \mathit{Alter}^{12\cdot100}_{2+12\cdot99,-1}(\mathit{day}, \mathit{Alter}^{12\cdot4}_{2+12\cdot3,1}(\mathit{day},$ `pseudomonth`$)))$.

The last definition takes leap days into account. *Alter* is the *alternating tick* operator and *Group* is the grouping operator. It is not necessary here to understand these operators. The point is that in a user friendly implementation of these operators an evaluator for arithmetic expressions like $2 + 12 \cdot 3$ is needed. This should be be no problem as long as the expressions are simple enough. For more complex temporal notions, however, the expressions become also more complex, and eventually a full size programming language is needed here. For example, for modelling ecclesiastical calendars, one needs to calculate the Easter date, and the algorithm for this is too complex to be expressed as a simple arithmetic formula.

The consequence for the PartLib library was to introduce a partitioning type *algorithmic partitionings*, which is specified by providing concrete algorithms in a concrete programming language (C++ in this case). Nevertheless, this is an exception. The general guideline is 'as algorithmic as necessary, as symbolic as possible'. The algorithmic partitionings can be used to define what some authors call *basic calendars* [31, 19].

**2. Separation of structure and meaning:**
An infinite sequence of non-overlapping granules is in principle also a partitioning of the time domain, if the gaps between the granules and within the granules are considered as part of the partitioning. Therefore one can turn a partitioning into a granularity by labelling certain partitions as gaps, and labelling the partitions which should belong to a granule with a common name. This has the advantages that the algorithms can be separated into a part which deals with the structure of the partitions, and a part which deals with the labels. Moreover, the labels can be used for other purposes. For example, the labels of a bus timetable can be the bus identifiers, and these can be keys for a bus database.

Notice that the notion of a 'label' in this chapter is different to the notion of a 'label' in the literature about granularities. Labels in this chapter are *names*, i.e. strings like 'Monday', 'Tuesday' etc. The 'labels' in the literature about granularities correspond to *coordinates* in this chapter.

PartLib provides no means for representing overlapping granularities as a single object. They must be represented as two separate partitionings.

**3. Compact data structures and efficient algorithms:**
The partitionings must be represented with finite data structures which support a number particular algorithms. This excludes certain problematic operators for constructing new partitionings (granularities) from existing ones. An example is the *union* operator. To understand this, consider a representation of, say, 'Tom's working day' and 'Jane's working day'. If Tom's working day is every day from 8 am until 5 pm, and 'Jane's working day' is every day from

9 am until 6 pm, then the union operation on these two partitionings is unproblematic. The resulting partitioning is easily representable in a compact way. If, however, Tom's job is to watch the moon in an observatory, then his working day may need to follow the moon phases, which can be represented with an algorithmic partitioning. The union of the two partitionings 'Tom's working day' and 'Jane's working day' is now a really complicated object, and not easily represented. Therefore we exclude operations like union, intersection etc. in PartLib itself. Instead we provide such operations in the WebCal system at another level. What is easy to realize is an operation which cuts 'Tom's working days' and 'Jane's working day' out of a given *finite* interval and then applies the set operation to the two intervals. Therefore 'Tom's and Jane's working day' would not be represented as a partitioning, but as a function that takes a time interval $I$ and returns the subintervals of $I$ which corresponds to the union of Tom's working days and Jane's working days. A prototype language for defining such functions is published in [37, 38]. The concrete language to be included in the WebCal system, however, has jet to be defined. The basic building blocks for it, however, are contained in [40, 39], and in this chapter.

**4. Intuitive specification of user defined partitionings:**
Most basic time units of calendar systems have a non-trivial algorithmic component. In WebCal they are therefore realized as built-in partitionings. Many others, however, are application specific or user defined. Therefore various authors have come up with algebraic operations for constructing new partitionings (granularities) from given ones [19]. The art is to find a basic set of operations which allows one to define new partitionings in a way which is intuitive to the user, and which provides good data structures for the algorithms. This set should be as small as possible in order to reduce the burden to develop the corresponding algorithms. On the other hand, it should be powerful and expressive enough that most real world examples for periodic temporal notions can be specified.

Besides the algorithmic partitionings, PartLib provides two more basic types of specifications: 'duration partitionings' and 'folded partitionings'. Duration partitionings are specified by an anchor time and a sequence of 'durations'. A *duration* is something like '1 month + 3 day', where 'month' and 'day' represent previously defined partitionings. 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'.

Notice that this specification is different to ('56 hour', '112 hour'). The difference is that when standard time changes to daylight savings time then the day is only 23 hours long, and when daylight savings time changes to standard time then the day is 25 hours long. A proper representation of 'day' as an algorithmic partitioning can take this into account, such that '8 hour + 2 day' would be the correct time shift even in this case. The specification of 'weekend' with a duration of '56 hour', however, would become wrong during the daylight savings time period.

PartLib provides three specializations of duration partitionings. They allow for faster algorithms, and they are more intuitive in certain cases. The first specialization, the *regular partitionings* covers the case that the durations are all of the same kind '$n$ $P$' where $P$ is always the same partitioning. For example, 'semester' could be defined this way. The anchor time is the start of, say, the winter semester. The durations are ('6 month', '6 month'). The first partition would be labelled 'winter semester', and the second partition would be labelled 'summer semester'. The algorithms for this simple kind of duration partitioning are more efficient than in the general case.

Another specialization are *date partitionings*. The partitions are in this version specified by concrete dates. In many countries, for example, people used to count the years from the beginning of the reigns of their emperors, and these are concrete dates. At a first glance, this specifies a partitioning of only a finite part of the time domain. This would require all the algorithms to check whether the time points under consideration are in the valid part of the time line, where the partitioning is specified, or not. With a simple trick, however, one can turn this finite partitioning into an infinite partitioning, and thus avoid these special cases. The trick is to turn the difference between two consecutive dates into durations. For example, the two dates 2004/5/10 and 2006/8/15 can be turned into a duration '2 year + 3 month + 5 day'. This way a date partitioning is turned into a duration partitioning. The finite part of the date partitioning is then automatically extrapolated into the infinite future and past. PartLib provides means to define boundaries for the partitionings, but these boundaries are not checked by the algorithms. It is up to the application of PartLib to check the boundaries.

Duration partitionings are the second basic type of partitionings. The third type are *folded partitionings*. Consider a bus timetable, which changes from season to season. The best way to specify this, would be to specify the seasons first, and for each season to specify the particular bus timetable. The 'folded partitioning' specification operation takes as input a *frame partitioning*, for example the seasons, and a sequence of *folded partitionings*, for example the four different bus timetables. It maps the folded partitionings automatically to the right frame partition, such that from the outside the whole thing looks like an ordinary partitioning.

**5. Support for certain key operations with partitionings:**
A very natural operation is to measure the distance between two time points in terms of a given time unit. 'The distance between $t_1$ and $t_2$ is 3.5 weeks', could, for example, be a useful information. Measuring the distance between two time points in terms of partitions of fixed length, for example seconds, is no point. It becomes more difficult if the time units have varying lengths. 'The distance between $t_1$ and $t_2$ is 3.5 months' is a nontrivial statement, because it depends on the location of $t_1$ and $t_2$ on the time line.

PartLib provides two *length* functions. The first one measure the distance between two time points in partitions of a given partitioning, and the second one measures the distance in granules. 'The distance between $t_1$ and $t_2$ is 1 working day', for example, is a possible outcome, even if 'working day' is defined not as a partitioning, but as a granule with a gap in it (for lunch time).

The second very natural operation is a shift operation, also in terms of partitions or granules. For example, one can ask a PartLib method to shift a time point $t$ by, say 3.5 months, or 3.5 working days into the future. Since the lengths of the partitions and granules may vary, the concrete amount, $t$ is shifted depends on the location of $t$ on the time line. It turns out that the notion of a time shift by some partitions is ambiguous. There are at least two different ways to do this, with more or less intuitive results. This problem is discussed in detail in Section 3.4.

PartLib offers some further operations. They are presented in the corresponding sections of this chapter.

After a brief review of PartLib's time domain we present the formal definitions of the PartLib's basic concepts and then discuss the specification mechanisms and the operations on partitionings. The interface to the PartLib implementation is presented in the appendix.

### Time Measurements and the Semantics of Computer Time

The backbone of our time representation is a reference time line, measured in seconds. The relation between the artificial counting of seconds in the computer and the real flow of time on our planet is determined by the physics of time measurement. Before the adoption of the UTC standard (Coordinated Universal Time) in 1972, a second was just the 86400th fraction of a day, measured between two subsequent zeniths of the sun. Since the rotation of the earth is not perfectly stable over the year, and, moreover, slows down from year to year, these seconds corresponded to varying time intervals. After the adoption of the UTC standard, a second corresponds to exactly 9.192.631.770 cycles of the light emitted when an electron jumps between the two lowest hyperfine levels of the Cesium 133 atom (measured and coordinated by the 'Bureau International des Poids et Mesures' in Paris, URL: http://www.bipm.fr/). The synchronization with the rotation of the earth is achieved by inserting a leap second almost every year by the International Earth Rotation Service (URL: http://hpiers.obspm.fr/). Therefore the seconds in our modelling of partitionings for the time before 1972 correspond to a fraction of the day. For the time after 1972 they correspond to the atomic seconds of the TAI standard (Temps Atomique International).

In this chapter it is assumed that there is a *global reference time GRT*, measured in seconds or some fraction of a second. *GRT* is actually isomorphic to the real numbers, but the number 0 corresponds to a particular point in time. As it is common in Unix systems, the origin of the reference time in our examples is the beginning of the year 1970 at the 0-meridian.

**Remark 3.1.1** *Since the real numbers $\mathbb{R}$ are used as the time axis, we speak of the* earliest *or* leftmost *number, time point or interval if we mean the one closest to $-\infty$. We speak of the* latest *or* rightmost *number, time point or interval if we mean the one closest to $+\infty$.* ■

## 3.2 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 considered in this chapter 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 natural numbers (their *coordinates*). For example, we could have the following enumeration

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

It is not by chance that half open intervals are used in this example. 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 chapter we therefore use the first type $[a, b[$.

Since all time measurements are done in discrete units (seconds, ticks of a clock, hyperfine transitions in a Cesium atom etc.) it makes sense to take integers as boundaries of the partitions. In the examples they represent seconds. Any other fraction of a second is possible as well.

Multiples of seconds are not possible without loosing precision because the leap seconds are ignored in this case.

The formal definition for partitionings of $\mathbb{R}$ which is used in this chapter is:

**Definition 3.2.1 (Partitioning)** *A partitioning $P$ of the real numbers $\mathbb{R}$ 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.* ■

Some useful notations for partitionings are defined:

**Definition 3.2.2 (Notations)** *Let $P$ be a partitioning.*

1. *For a partition $p = [s, t[$ in $P$ let $p_{[}\ s$ be left boundary of $p$ and let $p_{]}\ t$ be the right boundary of $p$.*
2. *For a time point $t$ and a partitioning $P$, let $t^P$ be the partition in $P$ which contains $t$.* ■

A sequence of finite partitions of the real numbers is in fact isomorphic to the integers. This can be exploited to give the partitions *addresses* or *coordinates*. The coordinates are very useful for navigating through sequences of partitions. Therefore we introduce *coordinate mappings*. In principle, there are many different coordinate mappings for a given partitioning, but for the intended application of the partitioning concept described in this chapter, one single coordinate mapping is sufficient. Therefore this unique coordinate mapping becomes an integral component of a partitioning.

**Definition 3.2.3 (Coordinate Mapping)** *A* coordinate mapping *of a partitioning $P$ is a bijective mapping between the intervals in $P$ and the integers. Since we usually use one single coordinate mapping for a partitioning $P$, we can just use $P$ itself to indicate the mapping.*

*Therefore let $p^P$ be the* coordinate *of the partition $p$ in $P$.*

*For a coordinate $i$ let $i^P$ be the partition which corresponds to $i$.*

*For a time point $t$ let $P.pc(t)\ (t^P)^P$ be the coordinate of the partition containing $t$. (pc stands for 'partition coordinate').*
*Let $P.sopT(t)\ {t^P}_{[}$ be the start of the partition containing $t$.*
*Let $P.eopT(t)\ {t^P}_{[}$ be the end of the partition containing $t$.*

*For a coordinate $i$ let $P.sopC(i)\ {i^P}_{[}$ be the start of the partition with coordinate $i$.*
*Let $P.eopC(i)\ {i^P}_{[}$ be the end of the partition with coordinate $i$.*

*For a time point $t$ we define*

$$P.lopT(t)\ P.eopT(t) - P.sopT(t)$$

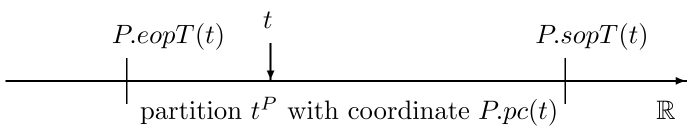
*as the length of the partition containing $t$.*

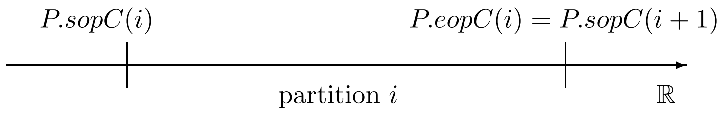
*For a coordinate $i$ we define*

$$P.lopC(i)\ P.lopT(P.sopC(i))$$

*as the length of the partition with coordinate $i$.* ■

The two pictures below illustrate the transitions between time points, coordinates and partitions:





**Example 3.2.4 (Seconds and Minutes)** *The partitioning for seconds consists of the sequence of intervals* $\ldots [-i, -i+1[ \ldots [0, 1[ \ldots [k, k+1[ \ldots$. *The interval* $[0, 1[$ *represents the first second in January* $1^{st}$ *1970, and its coordinate is 0.*
*The partitioning for minutes is*

| | | | 1970/1/1 ↓ | | | |
|---|---|---|---|---|---|---|
| *ref.time* : | ... | $[-60, 0[$ | $[0, 60[$ | $[60, 120[$ | $[120, 180[$ | ... |
| *coordinate* : | ... | $-1$ | 0 | 1 | 2 | , ... |

■

**Remarks:**
1. Partitions are not explictly represented in PartLib, only time points and coordinates. The functions which map time points to coordinates and back are therefore the important ones. Thus, the formulations of the algorithms below do not refer to partitions, but use these functions.

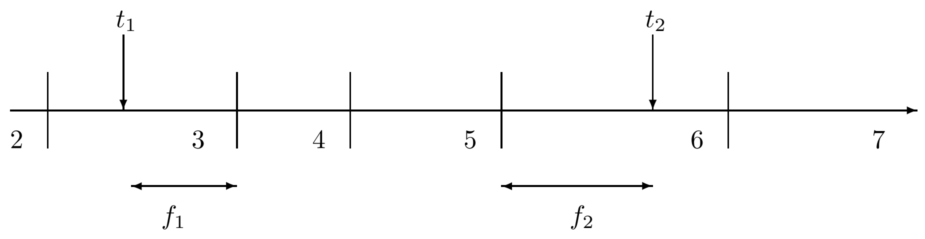
2. We use the notation $P.pc(t)$ for the function that maps a time point $t$ to the coordinate of its partition in the partitioning $P$. Alternative notations would be $pc(P, t)$ or $pc_P(t)$ or $pc^P(t)$. The notation $P.pc(t)$ comes from object oriented programming. $P$ is an object (instance of a class), and *pc* is a method in this class. This notation has two advantages. First of all, it is a bridge to the actual implementation where the program code looks just so. Secondly, it emphasizes the special role of $P$ as the context for the *pc* function as well as a number of other functions. Therefore we shall use the dot notation '*P.*' for most of the functions which depend on partitionings and other objects. If it is clear from the context, which object is meant, we may omit this object, and just use the function name.

## 3.2.1 Length of Intervals in 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 business trip and will be back in 3 months time'.

A very useful function is therefore $P.length(t_1, t_2)$, which measures the length of the distance between $t_1$ and $t_2$ in terms of partitions of the partitioning $P$. For example, $month.length(t_1, t_2)$ 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, + the relative length of $f_2$, measured as a fraction of the length of partition 6, + the number of partitions in between.

**Definition 3.2.5 (Length in Partitions)** *Let $P$ be a partitioning.*

*For two time points $t_1$ and $t_2$ with $t_1 \leq t_2$ we define*

$$P.lengthP(t_1,t_2) \begin{cases} \dfrac{t_2 - t_1}{P.lopT(t_1)} & \text{if } P.pc(t_1) = P.pc(t_2) \\ P.pc(t_2) - P.pc(t_1) - 1 + \\ \dfrac{P.eopT(t_1) - t_1}{P.lopT(t_1)} + \dfrac{t_2 - P.sopT(t_2)}{P.lopT(t_2)} & \text{otherwise} \end{cases}$$

*If $t_2 < t_1$ then $P.lengthP(t_1,t_2)$ $-P.lengthP(t_2,t_1)$.* ■

$P.lengthP(t_1,t_2)$ 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 $day.length(t_1,t_2) < 1$ even if $hour.length(t_1,t_2) = 24$. This happens when daylight savings time is disabled just during the interval $[t_1,t_2[$ and the day is 25 hours long.

**Definition 3.2.6 (modulo, remainder, $\lfloor ... \rfloor$ and $|\ldots|$)**
*The mod and remainder functions are used to map integers to non-negative indices $0,\ldots,n-1$. Therefore we need versions where the resulting values are between 0 and $n-1$, even for negative numbers. mod and remainder are defined for positive numbers as usual. For the negative numbers there are two different possibilities. We need the version where the resulting value is positive.*

*That means, $k$ mod $n$ is chosen such that for example $4$ mod $3 = 1$ and $-4$ mod $3 = 2$.*

*$m/n = k$ remainder $l$ is chosen such that for example $4/3 = 1$ remainder $1$ and $-4/3 = -2$ remainder $2$.*

*Let $\lfloor m \rfloor$ be the integer part of $m$ such that $\lfloor 3.5 \rfloor = 3$ and $\lfloor -3.5 \rfloor = -3$.*

*For an interval $s$ let $|s|$ be the length of the interval.* ■

### 3.2.2 Labels

For many periodic temporal notions there are standard names for the partitions. For example, days are named 'Monday', 'Tuesday' etc., months are named 'January', 'February' etc., seasons are named 'winter', 'spring' etc. If these names are attached as *labels* at the partitions, temporal notions like 'next summer' etc. can be modelled in an elegant way.

**Definition 3.2.7 (Labelled Partitionings)** *A* Labelling $L$ *is a finite sequence of labels (strings)* $l_0, \dots, l_{n-1}$.

*A labelling* $L = l_0, \dots, l_{n-1}$ *is turned into a* labelling function $L(i)$ *for a coordinate* $i$*:*

$$L(i)\ l_{i\ mod\ n}$$

■

A labelling $L$ can now be very easily attached to a partitioning: the partition with coordinate $i$ gets label $L(i)$.

**Example 3.2.8 (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*

$$L\ Th, Fr, Sa, Su, Mo, Tu, We.$$

*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,* $L(-1) = We$*, i.e. December 31 1969 was a Wednesday.* ■

In order to support internationalization, PartLib offers the possibility to use different names for the same label. The different names are distinguished by *realms*, which are also just strings. For example, one can define two realms 'en' and 'de' (for English and German). A label $l$ can now have the name 'Monday' in the realm 'en' and 'Montag' in the realm 'de'. This way all labels change their names when the realm is changed. Since for the rest of this chapter, this extra feature is not important, we can assume that labels are just strings.

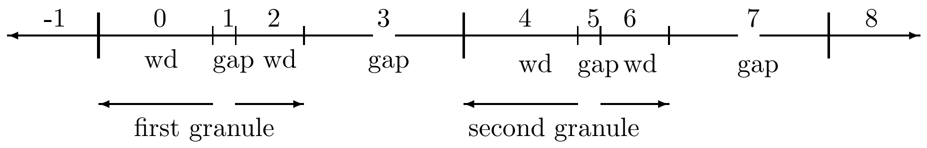
### 3.2.3 Granules

The label 'gap' is reserved for a special purpose. It can be used to denote gaps between semantically related partitions. For example, if the partitions represent school holidays then the periods between the school holidays can be named 'gap'. Gaps, together with the possibility to use the same label at different positions in a labelling makes it possible to define *granules*, with the same effect as in the original definitions of granularities. As an example, consider again the definition of 'working day' as a period of time between 8 o'clock am and 5 o'clock pm, interrupted by a lunch break between 1 o'clock pm and 2 o'clock pm. This could be defined in two stages. First, we define a partitioning with an anchor time at some particular Monday 8 o'clock am, and durations '5 hour', '1 hour', '3 hour', '16 hour'. The '1 hour' period is the lunch break and the '16 hour' period is the time between two 'working days'.

A suitable labelling is 'working_day, gap, working_day, gap'. A *granule* is characterized as a maximally long subsequence of a labelling such that the non-gap labels in the granule are the same. 'working_day, gap, working_day' in the above labelling is a granule, and the only granule.

Notice that we include gaps in this definition of a granule. Gaps are, however, excluded for most of the computations which involve granules.

If a labelling is attached at a partitioning, we get granules as subsets of the time line. The next picture illustrates this for the 'working_day' example. 'wd' stands for 'working_day'.



Only the partitions 0-2, 4-6 etc. represent granules. The partitions 2-4 are not a granule, although they are also labelled 'working_day, gap, working_day'. This is prevented because the definition of granule is based on the initial labelling, in this case 'working_day, gap, working_day, gap'.

**Definition 3.2.9 (Granule)** *Let $L = l_0, \ldots, l_{n-1}$ be a labelling. A* granule of the labelling *$L$ is a maximal subsequence $G = l_k \ldots l_m$ of $L$ such that*
*(i) $l_k \neq gap$ and $l_m \neq gap$, and*
*(ii) all non-gap labels in $G$ are the same.*

*A granule of a labelling function $L(i)$ (Def. 3.2.7) is an interval $[k, m]$ of coordinates, such that $l_{k \bmod n}, \ldots l_{m \bmod n}$ is a granule of the labelling $L$.*

■

A labelling like 'Monday, Tuesday, ...' without gaps and with the labels all being different has granules consisting of a single label each. A labelling may of course also have several non-trivial granules. To see this, let us take our 'working_day' example a bit further. The underlying partitioning partitions also the weekends. If we want to specify that 'working_days' are only from Monday till Friday, we could do this with the following labelling: 'wdMo, gap, wdMo, gap, wdTu, gap, wdTu, gap, wdWe, gap, wdWe, gap, wdTh, gap, wdTh, gap, wdFr, gap, wdFr, gap, gap, gap, gap, gap, gap, gap, gap, gap'. This labelling has five different granules 'wdMo, gap, wdMo', 'wdTu, gap, wdTu', 'wdWe, gap, wdWe', 'wdTh, gap, wdTh', 'wdFr, gap, wdFr', one for each day. The last 8 gaps exclude the weekend.

Each partition in a labelled granule can either be part of a granule or not be part of granule. If it is not part of a granule, then it is gap-labelled, but the neighbour non-gap labels are different.

A partitioning without an explicit labelling defines of course also granules: each partition is a granule consisting just of this single partition.

We define a function *closestGranule* which returns the coordinates of the closest granule. It comes in two versions. The version *closestGranuleC*$(i)$ takes a coordinate as argument, and the version *closestGranuleT*$(t)$ takes a time point as argument.

**Definition 3.2.10 (Closest Granule)** *Let $L$ be a labelling of a partitioning $P$, $i$ a coordinate and $t$ a time point.*

*The function $L$.closestGranuleC$(i)$ returns a pair $[k, m]$ of coordinates, such that*
*(i) $[k, m]$ is a granule of the labelling function $L(i)$, and*

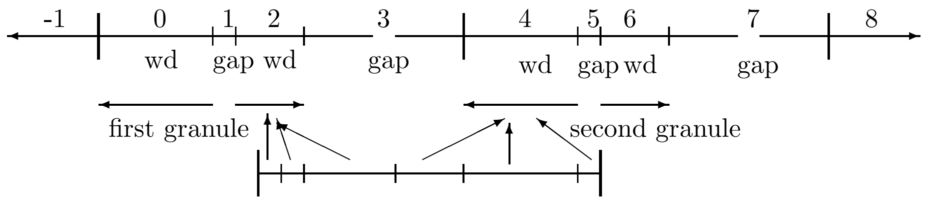
*(ii) either $i$ is within a granule, i.e. $k \leq i \leq m$, or $i$ is not within a granule, but it is closer to the granule $[k, m]$ than to any other granule. That means precisely, if $[a, b]$ are the coordinates of another granule, and $b \leq i \leq k$ then $k - i \leq i - b$, and if $m \leq i \leq a$ then $a - i \leq i - m$.*

*The function L.closestGranuleT$(t)$ returns a pair $[k, m]$ of coordinates, such that*
*(i) $[k, m]$ is a granule of the labelling function $L(i)$, and*
*(ii) either $t$ is within a granule, $i$ is not within a granule, but it is closer to the granule $[k, m]$ than to any other granule. That means precisely, if $[a, b]$ are the coordinates of another granule, and P.eopC$(b) \leq t \leq$ P.sopC$(k)$ then P.sopC$(k) - t \leq t -$ P.eopC$(b)$, and if P.eopC$(m) \leq i \leq$ P.sopC$(a)$ then P.sopC$(a) - t \leq t -$ P.eopC$(m)$.* ■

Notice that *closestGranule* prefers granules which lie in the future of $i$ or $t$. That means, if $i$ or $t$ is exactly in the middle between two granules then the future one is chosen.

**Length of intervals in granules:**
Once the notion of 'working_day' is defined by means of granules, it is quite natural to measure intervals in terms of the length of a 'working_day'. For example, you may want to say that 'these four hours are half a working day'. This is not much of a problem, if the granules all have the same length. If not, it depends on the position of the interval. 'Half a working day' may mean something different if I measure it on a Monday where I work 8 hours, or on a Friday, where I work, say, only 4 hours. Even worse, what if the interval lies on a weekend, and therefore outside any granules. It may still make sense to say that 'these four hours are half a working day', if I consider to shift the interval into a working day.

The function *lengthG*$(t_1, t_2)$ measures the distance between $t_1$ and $t_2$ in terms of granules. It deals with the problem that part of the interval $[t_1, t_2[$ or even the whole interval may lie outside any granule. The basic idea is to split the interval $[t_1, t_2[$ according to the given partitioning, determine for each subinterval the closest granule, and measure the subinterval in terms of this closest granule. The arrows in the next picture show which granules are used to measure the different parts of the interval.



**Definition 3.2.11 (Length in Granules)** *Let $P$ be a partitioning which is labelled with a labelling $L$. The function P.lengthG$(t_1, t_2)$ measures the distance between $t_1$ and $t_2$ in terms of granules as follows:*

*Step 1: The interval $[t_1, t_2[$ is partitioned into subintervals $s_1, \ldots$ such that the subinterval boundaries are aligned with the partition boundaries of $P$.*

*Step 2: The middle point $t_i$ for each subinterval $s_i$ is computed, and the coordinates $[l, m]$ of the granule closest to $t_i$ is determined by the closestGranuleT function (Def. 3.2.10). Let $l_i$ be the length of the non-gap partitions in this granule.*

*Step 3: The relative lengths $|s_i|/l_i$ are added together to give the result of P.lengthG$(t_1, t_2)$.* ■

The *lengthG* function yields intuitively correct results if the interval $[t_1, t_2[$ is a subset of the gap or non-gap partitions of a granule. For intervals lying outside any granule, it is questionable whether the closest granule is the right reference granule. It may always be the next future granule, or it may be a very particular reference granule. These options are not built-in in PartLib, but they can easily be programmed using its interface functions.

### 3.2.4 Relations Between Partitionings

PartLib supports four different relations between partitionings.

**Definition 3.2.12 (Relations Between Partitionings)** *Let $P$ and $Q$ be two partitionings. We define the following four relations between $P$ and $Q$.*

1. *$P$ has shorter partitions than $Q$ if the largest partition of $P$ is shorter than the shortest partition of $Q$ ($P$ is finer grained than $Q$.)*
2. *$P$ has shorter granules than $Q$ if the largest granule of $P$ is shorter than the shortest granule of $Q$.*
3. *$P$ includes the partitions of $Q$ if each partition of $P$ is a subset of a partition of $Q$.*
4. *$P$ includes the granules of $Q$ if each granule of $P$ is a subset of a granule of $Q$.* ■

These four relations are easy to define, but difficult to compute because for algorithmic partitionings it is not possible to compute the minimal or maximal partition length. Therefore PartLib uses an approximation. It generates random time points and computes the partition and granule length for these random points and certain points in their neighbourhood. How many points in the neighbourhood are to be checked is controlled by the 'repetition' parameter. For example, for the partitioning 'month' one would check 12 subsequent months for each random time point. This way, the 'local structure' of the partitionings is checked completely, whereas only finite samples check the 'global structure', in the 'month' example, the leap years.

The relations 'includes the partitions of' and 'includes the granules of' are also checked with finitely many randomly generated time points and their neighbourhoods.

## 3.3 Date Formats

In the subsequent section various notions are introduced in a recursive way: date formats, shifts, durations, and different types of partitionings.

We start with the definition of date formats.

**Definition 3.3.1 (Date Format)** *A* date format *$DF$ is a sequence $P_0/\dots/P_k$ of partitionings.*

*A* date *in a date format $DF$ is a sequence $d_0/\dots/d_n$ of integers with $n \leq k$.* ■

In principle, the date formats can consist of arbitrary partitionings. In most calendar systems there are, however, a few particular date formats. The Gregorian calendar, for example, has the two date formats year/month/day/hour/minute/second (where the names stand for the corresponding partitions), and year/week/day/hour/minute/second.

There are, however, two big differences between common date formats and the particular interpretation of the numbers in the dates we need in this chapter. Consider the date format year/month/day/hour/minute/second. The first difference is the interpretation of the year. The number 30, for example, for the ‘year’ part in the date format is the coordinate of the year. If the year 1970 has coordinate 0 then ‘30’ is the coordinate of the year 2000. The next difference has to do with the way we count months, weeks and days. Usually these are counted from 1. That means, January is month 1, the first week in a year is week 1, and Monday is day 1. In contrast to this, we count hours, minutes and seconds from 0. The first hour in a day is hour 0, the first minute in an hour is minute 0 etc. Our interpretation of date formats like the above is that months, days etc. denote shifts, instead of absolute values. In this interpretation the date 30/1 denotes a shift of 1 month from the beginning of the year with coordinate 30 (i.e. the year 2000). This is the beginning of February. Thus, all time units are counted from 0.

By interpreting the numbers as shifts, there is no problem to deal with arbitrary big numbers, and even with negative numbers. The date 30/200/-50, for example, in the date format year/month/day/hour/minute/second denotes a time point $t$ which is obtained from the beginning $s$ of the year 2000 (30 years after 1970), by shifting $s$ 200 months into the future, and from there 50 days into the past.

With these ideas in mind we can define a function *date*, which turns a time point into a date of the given format.

**Definition 3.3.2** (*date*) *Let $DF = P_0/\dots/P_k$ be a date format, and $t$ a time point.*
*Let $d_0/\dots/d_k$ $DF.date(t)$ where*
*$d_0$ $P_0.pc(t)$ and*
*$t_i$ $P_i.sopT(P_{i-1}.sopT(t))$ and*
*$d_i$ $\lfloor P_i.lengthP(t_i, t)\rfloor$ for $i = 1\dots k$.* ■

$d_0$ is the absolute coordinate of the $P_0$ partition containing $t$. $t_1$ is the starting point of the $P_1$ partition containing $t$. The $d_i$ are then calculated as $P_i$ increments compared to $t_i$ where $t_i$ is the beginning of the $p_i$ partition containing $t$.

For example, in the date format year/week/day/... $d_0$ is the coordinate of the year containing $t$. $t_1$ is the beginning of the first week overlapping with the year $d_0$. $d_1$ is then the integer part of the number of weeks between $t_1$ and $t$. $t_2$ is the beginning of the first day in the week containing $t$ etc.

The *date* function is exact only if the last partitioning $P_k$ corresponds to the integers of the time axis, usually seconds, or fractions of a second.

It is also possible to turn a date $d = d_0/\dots/d_n$ in a date format $DF = P_0/\dots/P_k$ into a time point. The function $TimePoint(d)$ gives the dates a precise semantics.

**Definition 3.3.3 (Time Point** *$DF.TimePoint(d)$***)** *Let $DF = P_0/\dots/P_k$ be a date format and let $d = d_0/\dots/d_n$ be a date in this format. The corresponding time point is defined:*
*$DF.TimePoint(d)$ $t_n$ where*
*$t_0$ $P_0.sopC(d_0)$ and*
*$t_i$ $P_i.sopC(P_i.pc(t_{i-1}) + d_i)$ for $i = 1\dots n$.* ■

In the date format year/week/day/..., for example, $t_0$ is the beginning of the year with coordinate $d_0$. *week.pc*$(t_0)$ is the coordinate of the week containing $t_0$. *week.pc*$(t_0) + d_1$ is the coordinate of the week which is $d_1$ weeks into the year. $t_1 = week.sopC(week.pc(t_0) + d_1)$ is the beginning of this week. $t_2$ is then the beginning of the day which is $d_2$ days into this week etc. Finally, $t_k$ is the coordinate of the second denoted by the given date.

By induction on the length of a date, one can prove that turning a time point $t$ into a date and the date back into a time point then we end up at the same time point $t$.

**Proposition 3.3.4** *For a date format DF whose last partitioning corresponds to the integers in the time axis, and a time point t:*

$$DF.TimePoint(DF.date(t)) = t.$$

■

The other direction, $DF.date(DF.TimePoint(d)) = d.$ need not hold, because there may be quite different dates which represent the same time point. For example, February 1st 2000 may be represented by 30/1/0 or by 30/0/31.

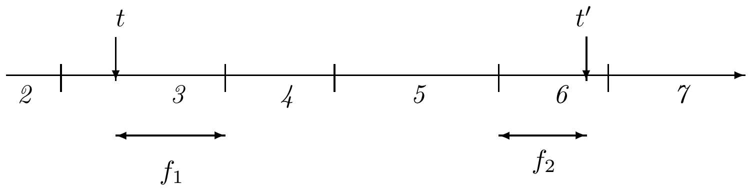
## 3.4 Shift Functions

Notions like 'in two weeks time' or 'three years from now' etc. denote time shifts. They can be realized by a function which maps a time point $t$ to a time point $t'$ such that $t' - t$ is just the required distance of 'two weeks' or 'three years' etc.

### 3.4.1 Length Oriented Shift Function

We define a function $P.shiftPL(t, m)$ which shifts a time point $t$ to a time point $t' = P.shiftPL(t, m)$ such that $P.lengthP(t' - t) = m$. ('shiftPL' stands for 'shift Partitions Length oriented', in contrast to 'shiftPD', which stands for 'shift Partitions Date oriented').

**Example 3.4.1 (for shiftPL)** *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 0.75 partitions, which is just 75% of the length of partition 6.* ■

**Definition 3.4.2 (shiftPL)** *Let P be a partitioning, t a time point, and m a real number. If $m \geq 0$ then*

$$shiftPL(t) \begin{cases} t + m * lopT(t) \quad \textit{if } t + m \cdot lopT(t) \leq eopT(t) \\ sopC(pc(t) + \lfloor m \rfloor) + (m - \lfloor m \rfloor) \cdot lopC(pc(t) + \lfloor m \rfloor) \quad \textit{if } sopT(t) = t \\ sopC(pc(t) + \lfloor m' \rfloor + 1) \\ + (m - m') \cdot lopC(pc(t) + \lfloor m' \rfloor + 1) \quad \textit{otherwise} \end{cases}$$

*where* $m'$ $m - (eopT(t) - t)/lopT(t)$.
*If* $m < 0$ *then*

$$shiftPL(t) \begin{cases} t + m * lopT(t) & \textit{if } sopT(t) \leq t + m \cdot lopT(t) \\ sopC(pc(t) + \lfloor m \rfloor) + (m - \lfloor m \rfloor) \cdot lopC(pc(t) + \lfloor m \rfloor - 1) & \textit{if } sopT(t) = t \\ sopC(pc(t) + \lfloor m' \rfloor) & \\ +(m - m') \cdot lopC(pc(t) + \lfloor m' \rfloor - 1) & \textit{otherwise} \end{cases}$$

*where* $m'$ $m + (t - sopT(t))/lopT(t)$.

■

It is not so difficult to see that the shiftPL function shifts a time point $t$ by $m$ partitions to a time point $t'$ such that the distance $t' - t$ is just $m$.

**Proposition 3.4.3 (Soundness of shiftPL)** *For any time point $t$:*
$shiftPL(t, m) - t = lengthP(t, shiftPL(t, m)) = m$
*Furthermore*
$shiftPL(shiftPL(t, m_1), m_2) = shiftPL(t, m_1 + m_2)$ ■

The proof is technical and gives no new insight. It is therefore omitted.

Unfortunately the shiftPL 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 above definition of shiftPL this is happens only by chance, or when the partitions have the same length.

### 3.4.2 Date Oriented Shift Function

PartLib provides another shift function, *shiftPD* 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 2004/1/15, then 'in one month time' usually means 2004/2/15. 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[1] in week 42 in the year 2004. In two weeks time would then be 2004/45/1.

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, 34/1/20/10/5/1. First we make a shift by three months and we end up at 34/4/20/10/5/1.

[1]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.

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 34/4/34/10/5/1. 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 34/4/34/22/5/1. This is turned back into a reference time.

This version of 'shift' gives more intuitive results. The drawback is that $shiftPD(t, m) - t = lengthP(t, shiftPD(t, m)) = m$ is usually no longer true. $shiftPD$ has in fact not much to do with $lengthP$.

The concrete definition of $shiftPD$ depends on the partitioning type. Therefore, we give them in the corresponding sections below (see Def. 3.7.5).

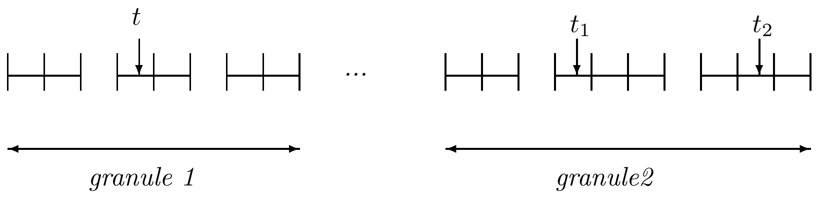
## 3.4.3 The Shift Function for Granules

A statement like 'we must move this task by three working days' refers to a shift of time points which is measured in granules. PartLib contains a 'shiftGranules' function, which does just this, it shifts time points by $m$ granules, where $m$ can be an integer or a fractional positive or negative value.

First of all, this function does not shift time points between two granules. I found no intuitive way to measure such a shift. The time point to be shifted must therefore be within a granule. It can, however, also be in a gap partition within a granule. The result may, however, in this case be quite unintuitive.

A formal definition of the algorithm, with all its special cases, is quite complex. Therefore we only explain it with an example, and omit the formal definition.

**Example 3.4.4 (for shifting granules)** *Suppose we want to shift the time point $t$ in the picture below by $m = 1.5$ granules.*



*Step 1: we determine the target granule into which the time point is to be shifted by moving from the granule containing $t$ $\lfloor m \rfloor$ granules forward. In this example, it is just one granule further.*

*Step 2: we determine the position $t_1$ in the target granule which corresponds to a shift of $\lfloor m \rfloor$ granules, in the example 1 granule. This is done in two steps:*

*Step 2a: we count the number of non-gap partitions from the start of the granule 1 until the partition containing $t$. In the example, $t$ is in the third non-gap partition. Therefore $t_1$ must be in the third non-gap partition of the target granule. Let us call this the target partition.*

*Step 2b: we measure the relative distance $d$ between the start of the partition containing $t$, and $t$ itself, and move $t_1$ forward from the start of the target partition by $d$ times the length of the target partition.*

*Step 3: in the example we must now move $t_1$ forward another 0.5 granule length. To this end we count the number $n$ of non-gap partitions in the target granule. In the example it is*

$n = 8$. $8 \cdot 0.5 = 4$. *Therefore we must move $t_1$ forward by 4 non-gap partitions. This can be done by first skipping 3 non-gap partitions and then moving $d$ times the length of the new target partition into it. We end up at position $t_2$.* ■

Shifts by negative granule numbers are analogous.

## 3.5 Durations

The partitionings are the mathematical model of periodic time units, such as years, months etc. This offers the possibility to define *durations*. A duration may for example be '3 months + 2 weeks'. Months and weeks are represented as partitionings, and 3 and 2 denote the number of partitions in these partitionings. The numbers need not be integers, but can be arbitrary real numbers.

A duration can be interpreted as the length of an interval. In this case the numbers should not be negative. A duration, however, can also be interpreted as a time shift. In this interpretation negative numbers make perfectly sense. $d = (-2\ week), (-3\ month)$, for example, denotes a backward shift of 2 weeks followed by a backward shift of 3 months.

**Definition 3.5.1 (Duration)** *A* duration $d = (d_0, P_0), \ldots, (d_k, P_k)$ *is a list of pairs where the $d_i$ are real numbers and the $P_i$ are partitionings.*

*If a duration is interpreted as a shift of a time point, it may be necessary to turn the shift around, in the backwards direction. Therefore the inverse of a duration is defined:*
$-d\ (-d_k, P_k), \ldots, (-d_0, P_0)$ ■

For example, if $d = (3\ month), (2\ week)$ then $-d = (-2\ week), (-3\ month)$.

In Def. 3.7.5 below a `shift` function for algorithmic partitionings is introduced. It shifts a time point by a number of partitions in a given partitioning. Any such shift function can be lifted to operate on durations.

**Definition 3.5.2 (`shift` for Durations)** *Given a function $P.shift(t, m)$, which shifts a time point $t$ by $m$ partitions of the partitioning $P$, we define a corresponding* `shift` *function for durations:*

$$D.shift(t)\ \ P_k.shift(\ldots P_1.shift(P_0.shift(t, d_0), d_1)\ldots, d_k)$$

*where $t$ is a time point and $D = (d_0, P_0), \ldots, (d_k, P_k)$ is a duration.* ■

For example, if $D = (3\ month), (2\ week)$ then

$$D.shift(t) = week.shift(month.shift(t, 3), 2),$$

i.e. $t$ is shifted by 3 months first, and the resulting time point is then shifted by 2 weeks.

## 3.6 Global and Local Reference Time

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 and time zones. 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 and time zones.

The transition from GRT to LRT is done in two steps. The first step deals with the leap seconds and the second step deals with time zones. The transition from LRT to GRT goes the other way round.

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 3.6.1 (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 3.6.2 (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.* ∎

Time zones are characterized by an offset between the local time and the time at the 0-meridian. For example, if at the 0-meridian it is 0 o'clock then the offset to the time zone of Germany is 1 hour, i.e. in Germany it is already 1 am. The time zone offset is in this case +3600 seconds, and the local reference time is 3600 seconds ahead of the global reference time.

**Definition 3.6.3 (Transition between GRT and LRT)** *Given the correction functions $lsG$ and $lsL$ for leap seconds (Def. 3.6.1) and a time zone offset $tzo$, we define*

$$LRT(t)\ \ t - lsG(t) + tzo.$$

*for a global reference time $t$. $LRT(t)$ computes the local reference time from the global reference time.*

*The function*

$$GRT(t)\ \ t - tzo + lsL(t)$$

*computes the global reference time from the local reference time.* ∎

## 3.7 Specification of Partitionings

A partitioning is usually an infinite sequence of intervals, and these intervals need not be of the same length. Therefore it is not possible to specify such a partitioning by just enumerating its partitions.

Three basically different ways to specify partitionings are presented, and it is shown how a specification corresponds to a partitioning and an associated coordinate mapping. Since partitions themselves are not explicitly represented in PartLib, but only time points and coordinates, it is sufficient to give for each type of specification of a partitioning $P$ corresponding definitions of the 'start of partition' function $sopC(i)$ and the 'partition coordinate' function $pc(t)$. $sopC(i)$ maps a coordinate $i$ to the starting point of the corresponding partition and $pc(t)$ maps a time point $t$ to the coordinate of the partition containing $t$. The two functions are fundamental for various service functions.

The 'start of partition' function determines the partitioning completely because

$$p_i \;\; [sopC(i), sopC(i+1)[.$$

The 'partition coordinate' function $pc(t)$ is then derivable:

$$P.pc(t) \;\; \min_i (P.sopC(i) \leq t < P.sopC(i)). \tag{3.1}$$

This way, however, $pc(t)$ can only be computed through search, which is extremely inefficient. Therefore we give a more efficient definition of $pc(t)$ in each case. In most cases the algorithm for $pc(t)$ tries at first a good guess $i'$ for the coordinate, and then searches in the neighbourhood of $i'$ until the condition (3.1) is satisfied.

### 3.7.1 Algorithmic Partitionings

The first type of partitionings is mainly used for modelling the basic time units of calendar systems, years, months etc. The specification consists of an average length of the partitions, a correction function and an offset against time point 0.

**Definition 3.7.1 (Specification of Algorithmic Partitionings)**
*Algorithmic partitionings are specified by a the components $(avl, po, cf, DF)$ 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 a correction function, and*
4. *DF is a date format. The date format is needed for the $shiftPD$ 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 3.7.2 (The Function *sopC* for Algorithmic Partitionings)**
*The algorithmic partitioning $P$ which is specified by the triple $(avl, po, cf)$ (Def. 3.7.1) has the following 'start of partition' function:*

$$P.sopC(i) \;\; GRT(i \cdot avl + cf(i) + po). \qquad ■$$

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.

**Example 3.7.3 (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:* $\lambda(n)0$.

**minute:** *average length: 60, offset: 0, correction function:* $\lambda(n)0$.

**hour:** *average length: 3600, offset: 0, correction function:* $\lambda(n)0$.

**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 $pc(t)$ maps a reference time point $t$ to the coordinate of the partition containing $t$. For algorithmic partitionings this function is more complicated than $sopC(i)$ 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 $sopC(i)$ 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 3.7.4 (The Function $pc(t)$ for Algorithmic Partitionings)** *Let* $t$ *be a local reference time point for the given partitioning* $P = (avl, po, cf)$. *The algorithm for* $pc(i)$ *starts with a first guess* $i$ $t/avl$ *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*[2]*: Let* $r$ $sopC(i)$. *If* $r \geq t$ *let* $r'$ $sopC(i-1)$ *and compute* $i'$ $(r-t)/(r-r')$ *to get a better estimate* $i$ $i - i'$ *for the correct coordinate. If* $r < t$, $i$ *is increased in a similar way*[3].

*The second phase of the algorithm is simpler: the correct coordinate is searched by just decreasing or increasing* $i$ *by 1, until* $sopC(i) \leq t < sopC(i+1)$ *holds. The result of the function* $pc(t)$ *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 coordinate which was computed in the first phase. This phase guarantees that the result is

[2]The threshold in the implementation is 3. The initial value for $i'$ can be any number greater than the threshold. $i' = 10$ is fine.

[3]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.

correct, i.e. it satisfies the condition (3.1) for $pc(t)$. The algorithm is very efficient even if the average length of the partitions is quite different to their individual length.

We can now define the date oriented shift function for algorithmic partitionings. The idea of it was already explained in Section 3.4.2.

**Definition 3.7.5 (Date Oriented `shiftPD` for Algorithmic Partitionings)** *The function $P.shiftPD(t, m)$ where $t$ is a GRT time point, $m$ is a real number and $P$ is an algorithmic partitioning (sec. 3.7.1) with date format $DF = P_0, \dots$ performs the following steps:*

```
1. Let $d = d_1/\dots/d_k$ $DF.date(t)$ (Def. 3.3.2);
2. $i$, $d$ and $m$ are now modified destructively:
   while($m \neq 0$ and $i \leq k$)
     (a) $d_i$ $d_i + \lfloor m \rfloor$
     (b) if($i < k$)
           $t$ $DF.TimePoint(d)$        (Def. 3.3.3)
           $m$ $(m - \lfloor m \rfloor) \cdot P_{i+1}.lengthP(P_i.sopT(t), P_i.eopT(t))$
           $i$ $i + 1$.
3. the result of $P.shiftPD(t, m)$ is now $DF.TimePoint(d)$.
```
■

Although the shiftPD function gives intuitive results in most cases, it has a number of drawbacks. One of them was already mentioned: shiftPD has not much to do with the lengthP function. Measuring the shifted distance with the lengthP function does usually not give the expected results.

Another drawback is that shifting a time point $t$ first by $m_1$ partitions, and then by $m_2$ partitions may not be the same as shifting $t$ by $m_1 + m_2$ partitions. This holds only if $m_1$ and $m_2$ are integers. A counter example for the case that $m_1$ and $m_2$ are factional values is:

**Example 3.7.6 (Counterexample)** *Suppose we want to shift the time point 0 twice by 1.5 months. The date for 0 is 0/0/0. Since February 1970 has 28 days, a shift by 1.5 months ends up at 0/1/14. Another shift by 1 month yields 0/2/14. This is in March. March has 31 days. Therefore a shift by 0.5 months means a shift by 15.5 days. We end up at 0/2/29/12. This is different to the result of a direct shift by 3 months: 0/3/0.*

Nevertheless, the shiftPD is usually preferable. A striking example which illustrates the difference between shiftPD and shiftPL is such a simple thing as a shift by 1 day. If it is 5 pm, a shift by 1 day should end up at 5 pm next day. This is the case with the shiftPD function, even if during the night, standard time has been changed to daylight savings time. In contrast, the shiftPL function yields a very odd result in this case.

Other periodic temporal notions which can be modelled by algorithmic partitionings are, for example, sunrises and sunsets, moon phases, the church year which starts with Easter, etc. The specification for the western version of Easter would be: average length: 31536000 (1 year), offset: 7516800 (1970/3/29) and a correction function which actually computes the precise date of Easter (see for example [16]). The specification of all these partitionings must be accompanied by an appropriate `shiftPD` function.

The specification of the algorithmic partitionings requires the correction function $cf(i)$, and this is a piece of code. Therefore algorithmic partitionings are usually hard coded in the application program. This is different for the remaining three partitioning types. They can be specified purely symbolically. Therefore one can read their specification from a file or a database at run time. This makes the system very flexible.

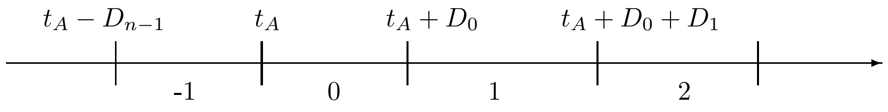
### 3.7.2 Duration Partitionings

Duration partitionings 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', and the second interval would be labelled 'gap'.

**Definition 3.7.7 (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. 3.5.1).* ■

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 not much of a chance to realize a 'start of partition' function *sopC* other than by just looping $i$ times over $D_0 \dots D_{n-1}$.

**Definition 3.7.8 (The Function *sopC* for Duration Partitionings)**
*The duration partitioning* $P$ *which is specified by the data* $(t_A, (D_0 \dots D_{n-1}))$ *(Def. 3.7.7) has the following 'start of partition' function:*

$$sopC(i) \;\; t_i$$

*where* $t_i$ *is determined by shifting the anchor time point* $t_A$ $i$ *times:*
*Let* $t_0 \;\; t_A$.
*if(*$i \geq 0$*): for* $j = 1, \dots, i$*:* $t_j \;\; D_{(j-1) \ mod \ n}.shift(t_{j-1})$. *(Def. 3.5.2)*
*if(*$i < 0$*): for* $j = 1, \dots, -i$*:* $t_j \;\; -D_{(n-j) \ mod \ n}.shift(t_{j-1})$.
■

Notice that the shift function for durations uses the shiftPD function (Sec. 3.4.1). Because two shifts by $m_1$ and $m_2$ partitions are not necessarily the same as one shift by $m_1 + m_2$ partitions (Ex. 3.7.6) , this has an effect on the meaning of duration partitionings. A duration partitioning with a duration '1.5 month + 1.5 month' is not the same as a duration partitioning with a duration '3 month'.

Because there is no further assumption about the durations in the specification of duration partitionings, there is not much chance to optimize the ‘partition coordinate’ function *pc* either. Therefore the definition (3.1) is taken as algorithm for *pc*.

The `shiftPD` function cannot be optimized either. It also loops over the duration $D_0 \dots D_{n-1}$ and calls the shift function for durations as often as necessary.

**Definition 3.7.9 (`shiftPD` for Duration Partitionings)**
*Let $P = (t_A, (D_0 \dots D_{n-1}))$ be a duration partitioning. The $P.shiftPD(t, m)$ function for a time point $t$ and a real number $m$ performs the following steps:*

```
    Let (k remainder l) m/n.
If $m \geq 0$:
    if $(m \geq 1)$: for $i = 0, \dots, \lfloor m \rfloor - 1$ let $t$ $D_{i \ mod \ n}.shift(t)$
    let $t$ $t + (m - \lfloor m \rfloor) \cdot (D_{\lfloor m \rfloor \ mod \ n}).shift(t) - t)$.
If $m < 0$:
    for $i = 0, \dots, -\lfloor m \rfloor - 1$ let $t$ $-D_{n-1-(i \ mod \ n)}.shift(t)$
    let $t$ $t + (m - \lfloor m \rfloor) \cdot (t - D_{(\lfloor m \rfloor \ mod \ n)}.shift(t)$.
    The result of $P.shift(t, m)$ is now the modified $t$.
```
■

## 3.7.3 Regular Partitionings

A special case of a duration partitioning is when all durations are of the form ‘$n$ $P$’ and the partitioning $P$ is the same in all durations. For this case there are more efficient ways than looping over lists of durations. Therefore, and because many partitionings are of this type, it makes sense to treat them in a special way.

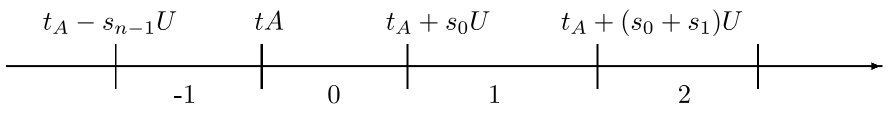
A typical example is the notion of a semester at a university. In the Munich case, the dates could be: anchor time: 970358431 (October 2000). The shifts are: 6 months (with label ‘winter semester’) and 6 months (with label ‘summer semester’). This defines a partitioning with partition 0 starting at the anchor time, and then extending into the past and the future. The partition with coordinate 0 in this example is the winter semester 2000/2001.

**Definition 3.7.10 (Specification of Regular Partitionings)** *A regular partitioning is specified by the triple $(t_A, U, (s_0 \dots s_{n-1}))$ where*

1. *$t_A$ is the anchor time point (in the global reference time)*
2. *$U$ is a partitioning, the time unit for the shifts,*
3. *$s_0 \dots s_{n-1}$ is a list of real numbers, the shifts.*

■

The partitions of regular partitionings are obtained by shifting the anchor point $t_A$ first by $s_0$ time units $U$, and then by $s + 0 + s_1$ time units $U$ etc.



This picture illustrates a subtle, but important difference to duration partitionings. The partition boundaries for duration partitions are computed by successively applying the shift function for the corresponding durations. The shift function for durations uses internally the date oriented shiftPD function, for which $shiftPD(shiftPD(t, m_1), m_2) \neq shiftPD(t, m_1 + m_2)$ may be the case.

In contrast to this the partition boundaries for regular partitions are computed by first adding the shifts together, and then shifting the anchor time in one single step. Both versions yield the same results for integer shifts. This is not guaranteed if the shifts are fractional.

**Definition 3.7.11 ()** *The regular partitioning $P$ which is specified by the data $(t_A, U, (s_0 \dots s_{n-1}))$ (Def. 3.7.10) has the following 'start of partition' function:*

$$sopC(i)\ \ U.shiftPD(t_A, s(i))$$

*where* $s(i)\ \ k \cdot \Sigma_{j=0}^{n-1} s_j + \begin{cases} \Sigma_{j=0}^{l} s_j & \text{if } i \geq 0 \\ \Sigma_{j=0}^{l-1} s_j & \text{otherwise} \end{cases}$

*and* $(k \text{ remainder } l)\ \ i/n$.

■

**Example 3.7.12** *Let the anchor date be 2000/1, and let the shifts be 3,4,5 months.*

*$sopC(5)$ is calculated as follows:*
$5/3 = 1$ *remainder* $2$.
$i' = 1 \cdot (3 + 4 + 5) + (3 + 4) = 19$.
*This means 2000/1 is to be shifted by 19 month, and we end up at the beginning of 2001/7.*

*$sopC(-5)$ is calculated as follows:*
$-5/3 = -2$ *remainder* $1$.
$i' = -2 \cdot (3 + 4 + 5) + 3 = -21$.
*This means 2000/1 is to be shifted by -21 month, and we end up at the beginning of 1998/3.* ■

The 'partition coordinate' function $pc$ turns the reference time into a coordinate $i$ of the time unit $U$ and then uses the difference between $i$ and the coordinate of the anchor time to compute the number of shifts which are necessary to get from the anchor time to the reference time. This is the coordinate of the partition containing $t$.

**Definition 3.7.13 (The Function $pc$ for Regular Partitionings)**
*Let $(t_A, U, (s_0 \dots s_{n-1}))$ be the specification of a regular partitioning. Let $t$ be a local reference time point.*
*Let $i\ \ U.pc(t) - U.pc(t_A)$ and $(k \text{ remainder } l)\ \ i/\Sigma_{j=0}^{n-1} s_j$.*

*Let $P.pc(t)\ \ k \cdot n + \max_{i \geq 0}((\Sigma_{j=0}^{i} s_j) \leq l)$.* ■

**Example 3.7.14** *Let the anchor date be 2000/1, and let the shifts be 3.5,4.5,5.5 months.*
*Let $t$ be such that $i\ \ U.pc(t) - U.pc(t_A) = 21$.*
*$k$ remainder $l = i/(3.5 + 4.5 + 5.5) = 1$ remainder $7.5$.*
*$P.pc(t) = 1 \cdot 3 + 1 = 4$. This is the partition between month 17 and 21.5.*

*Now let $t$ be such that $i\ \ U.pc(t) - U.pc(t_A) = -21$.*
*$k$ remainder $l = -21/13.5 = -2$ remainder $8$.*
*$P.pc(t) = -2 \cdot 3 + 2 = -4$.*
*This is the partition between month -23.5 and -19.* ■

**shiftPD:**
The `shift` function for regular partitionings is explained informally with the following example: Suppose we have a specification of *trimesters* in the following way: Anchor date: 2000/10, trimesters: 3,4,5 months.

We want to shift a time point $t$ by 8.5 trimesters. The following steps are necessary

1. $8/3 = 2$ *remainder* 2. That means, first a shift of 2 full cycles of $3+4+5 = 12$ months is performed, and we end up at a time point $t_1$.
2. The index $i'$ of the partition containing $t_1$ is computed, suppose it is 1, i.e. $t_1$ lies in the first trimester. In order to get two trimesters further, a shift of $3 + 4 = 7$ months is necessary, and we end up at time point $t_2$ which is in the third trimester.
3. The third trimester has 5 month. $5 \cdot 0.5 = 2.5$, i.e. we shift $t_2$ by another 2.5 month.

**Definition 3.7.15 (`shiftPD` for Regular Partitionings)**
*Let* $P = (t_A, U, (s_0, \ldots, s_{n-1}))$ *be a regular partitioning (Def. 3.7.10). The function* $P.shiftPD(t, m)$ *performs the following steps:*

1. *Let* $(k$ *remainder* $l)$ $\lfloor m \rfloor / n$
   *Let* $t_1$ $U.shiftPD(t, k \cdot \Sigma_{j=0}^{n-1} s_j)$.
2. *Let* $i'$ $t_1^P$.
   *Let* $t_2$ $U.shiftPD(t, \Sigma_{j=0}^{l-1} s_{(i'+j) \ mod \ n})$.
3. *Return* $U.shiftPD(t_2, (m - \lfloor m \rfloor) \cdot s_{(i'+l) \ mod \ n})$.

■

Further examples for periodic temporal notions which can be encoded as regular partitionings are decades, centuries, the British financial year which starts April $1^{st}$, the dates of a particular lecture, which is every week at the same time etc.

### 3.7.4 Date Partitionings

Date Partitionings are specified by providing the boundaries of the partitions as concrete dates.

An example could be the dates of the Time conferences: 1994/5/4 Time94 1994/5/5 gap 1995/4/26 Time95 1995/4/27 gap 1996/5/19 Time96 1996/5/21 gap 1997/5/10 Time97 1997/5/12 gap 1998/5/16 Time98 1998/5/18 gap 1999/5/1 Time99 1999/5/3 gap 2000/7/7 Time00 2000/7/10 gap 2001/6/14 Time01 2001/6/17 gap 2002/7/7 Time02 2002/7/10 gap 2003/7/8 Time03 2003/7/11 gap 2004/7/1 Time04 2004/7/4.

Another example could be the seasons: 2000/3/21 spring 2000/6/21 summer 2000/9/23 autumn 2000/12/21 winter 2001/3/21.

In the introduction I explained the trick how to turn these finitely many dates into an infinite partitioning: the differences between two subsequent dates are turned into durations. The durations are then used to extrapolate the partitioning into the infinity.

The seasons example above shows that this makes really sense. The time difference between the dates can be expressed as durations '3 month' for spring, '3 month + 2 day' for summer, '3 month - 2 day' for autumn and '3 month' for winter. The extrapolation of this now yields the seasons for the whole time line. This works even for the leap years, in which winter is one day longer. This is covered by the duration '3 month' for winter, which is one day longer in leap years.

**Definition 3.7.16 (Specification of Date Partitionings)** *A date partitioning is specified as a list dates $d_0, \ldots, d_n$ in a date format $DF$.* ■

The dates can very easily be turned into durations: Suppose the date format is $DF\ P_0, \ldots$. The difference between two dates $d_i = d_{i,0}/d_{i,1}/\ldots$ and $d_{i+1} = d_{i+1,0}/d_{i+1,1}/\ldots$ yields the following duration: $(d_{i+1,0} - d_{i,0}, P_0)$, $(d_{i+1,1} - d_{i,1}, P_1), \ldots$, and this is sufficient to specify a duration partitioning. The anchor time is given by $DF.timePoint(d_0)$.

Notice that the coordinate calculation for duration partitionings which applies the shift function for durations, and this uses the date oriented shift function for partitionings, undos the above subtractions. Therefore it reconstructs exactly the original dates.

### 3.7.5 Folded Partitionings

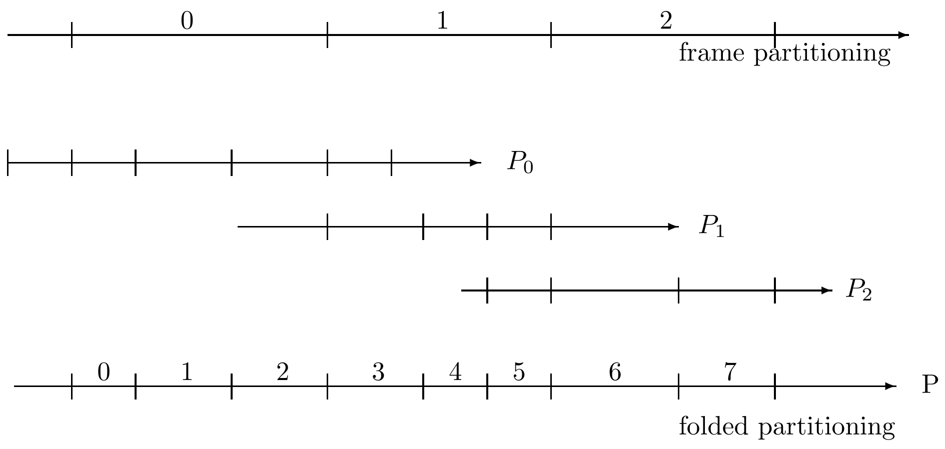
Another basic type of partitionings are *folded partitionings*. We explained already the bus timetable example. The bus timetable changes from season to season. The best way to specify this, would be to specify the seasons first, and for each season to specify the particular bus timetable. The 'folded partitioning' specification operation takes as input a *frame partitioning*, for example the seasons, and a sequence of *component partitionings*, for example the four different bus timetables. It maps the component partitionings automatically to the right frame partition, such that from the outside the whole thing looks like an ordinary partitioning.

**Definition 3.7.17 (Folded Partitioning)** *A folded partitioning $P$ is specified by a* frame partitioning *$F$ and a finite list $P_0, \ldots, P_{n-1}$ of* component partitionings*. The component partitionings must meet the following condition:*
*The partitions of the component partitionings must be aligned with the partitions of the frame partition. That means, for $i = 0, \ldots, n-1$, the start of each frame partition must be the start of a component partition in $P_i$, and the end of each frame partition must be the end of a component partition in $P_i$;*

*A folded partitioning is called* constant *iff each frame partition with coordinate $i$ contains the same number of component partitions as the frame partition with coordinate $i mod n$.* ■

The condition is in principle not necessary, but if it is not met, it complicates the algorithms enormously. It excludes, for example, that 'week' can be used as component partitioning when the frame partition is 'year'. 'month', however, can be used because years start and end with a month.

The figure below gives an idea how the coordinates for the folded partitioning are obtained from the coordinates of the frame partitioning and the component partitionings.



The algorithms for the 'partition coordinate' function *pc* and for the 'start of partition' function *sopC* are much more efficient if the partitioning is constant, i.e. the number of component partitions per frame partition of each component partitioning $P_i$ does not change over time. In this case it is possible to compute the total number of component partitions up to the beginning of a given frame partition by a few multiplications and additions. If the partitioning is not constant, one must iterate through all frame partitions and add the corresponding numbers together.

The bus timetable example is typical for a folded partitioning which is *not* constant. This is due to the leap years. Since the winter is one day longer in a leap year, the bus timetable in this winter has more partitions than in the other winters. An example for a constant folded partitioning could be the definition of working hours with shifts. In the first week I may have a morning shift, in the second week an evening shift, and in the third week a night shift. The number of partitions labelled 'working hour' and 'gap' would not change in this case.

The 'partition coordinate' function *pc* needs an auxiliary function *partitionsUpTo*, which computes for a frame coordinate the number of partitions up to the beginning of this frame partition. More precisely: for a positive frame coordinate $frco$, *partitionsUpTo* computes the number of component partitions from frame partition 0 (inclusive) up to the beginning of frame partition with coordinate $frco$, and for negative frame coordinates, it computes the *negated* number of component partitions from frame partition -1 (inclusive) down to and including the frame partition with coordinate $frco$.

We define this function for constant folded partitionings and for non-constant folded partitionings separately.

**Definition 3.7.18 (*partitionsUpTo* for constant folded partitionings)** *Let $P$ be a constant folded partitioning with component partitionings $P_0 \ldots, P_{n-1}$.*

*We define the function partitionsUpTo($frco$) where $frco$ is a coordinate in the frame partitioning as follows:*

*For $i = 0, \ldots, n-1$ let partitions($i$) be the number of component partitions per frame partition $P_i$. Since the partitioning is constant, it is sufficient to compute these numbers for the first n frame partitions.*

*Let cycle $\Sigma_{i=0}^{n-1} partitions(i)$.*

*Let $frco/n = blocks$ remainder rest.*

*Now we have:*

$partitionsUpTo(frco)\ blocks * cycle + \Sigma_{i=0}^{rest-1} partitions(i)$. ■

**Definition 3.7.19** (*partitionsUpTo* **for normal folded partitionings)** *Let $P$ be a folded partitioning with frame partitioning $F$ and component partitionings $P_0 \ldots, P_{n-1}$.*

*We define the function partitionsUpTo($frco$) where $frco$ is a coordinate in the frame partitioning as follows:*

*If $frco \geq 0$ let $from$ $0$ and $to$ $frco - 1$.*
*If $frco < 0$ let $from$ $frco + 1$ and $to$ $-1$.*

*Let $n$ $\Sigma_{i=from}^{to} P_{i \ mod \ n}.pc(F.eopC(i)) - P_{i \ mod \ n}.pc(F.sopC(i))$.*
*If ($frco \geq 0$) let partitionsUpTo($frco$) $n$.*
*If ($frco < 0$) let partitionsUpTo($frco$) $-n$.* ■

Now we can define the 'partition coordinate' function $pc(t)$. It uses the *partitionsUpTo* function to get the number of partitions up to the frame partition containing $t$, and then computes the remaining partitions locally with the corresponding component partitioning.

**Definition 3.7.20 (The Function** *pc* **for Folded Partitionings)** *Let $P$ be a folded partitioning with frame partitioning $F$ and component partitionings $P_0, \ldots, P_{n-1}$. Let $t$ be a global reference time point.*

*Let $frco$ $F.pc(t)$ be the frame coordinate for time point $t$.*
*We have:*
*$pc(t)$ $partitionsUpTo(frco) + P_{frco \ mod \ n}.pc(t) - P_{frco \ mod \ n}.pc(F.sopC(frco))$* ■

The 'start of partition' function *sopC* needs an auxiliary function $frameCoordinate(co) = (frco, rist)$, which computes for a given coordinate of a partition (i) the coordinate of the frame partition which contains this partition and (ii) the coordinate of the first component partition within this frame partition. There are again different definitions for constant folded partitionings and for normal folded partitionings.

**Definition 3.7.21** (*frameCoordinate* **for Constant Folded Partitionings)** *Let $P$ be a constant folded partitioning with component partitionings $P_0 \ldots, P_{n-1}$.*

*We define the function $frameCoordinate(co)$ where co is a $P$-coordinate as follows:*

*For $i = 0, \ldots, n-1$ let $partitions(i)$ be the number of component partitions per frame partition $P_i$.*

*Let cycle $\Sigma_{i=0}^{n-1} partitions(i)$.*

*Let $co/cycle = blocks$ remainder $rest$*

*Let $frcoblocks{\cdot}n$ be the coordinate of the first frame partition $p_0$ in the partitions $p_0, \ldots, p_{n-1}$ which correspond to $P_0, \ldots, P_{n-1}$, and where one of the $p_i$ contains the component partition with coordinate co.*

*Let $i$ $max_i(\Sigma_{k=0}^{i} partitions(k) \leq rest)$ be the offset from $frco$, such that $frco + i$ is the coordinate of the frame partition containing the component partition with coordinate co.*
*This way we get*
*$frameCoordinate(co)$ $(frco + i, blocks \cdot cycle + \Sigma_{k=0}^{i} partitions(k))$* ■

**Definition 3.7.22** (*frameCoordinate* **for Normal Folded Partitionings)** *Let $P$ be a constant folded partitioning with frame partitioning $F$ and component partitionings $P_0 \ldots, P_{n-1}$.*

*We define the function $frameCoordinate(co)$, where co is a $P$-coordinate as follows:*
*If $co \geq 0$ let*
*$frco$ $max_i(\Sigma_{k=0}^{i} P_{k \ mod \ n}.pc(F.eopC(k)) - P_{k \ mod \ n}.pc(F.sopC(k)) \leq co)$*

$\quad first\ \Sigma_{k=0}^{frco} P_{k \ mod \ n}.pc(F.eopC(k)) - P_{k \ mod \ n}.pc(F.sopC(k)) \leq co)$

*If* $co < 0$ *let*

$\quad frco\ - max_i(\Sigma_{k=1}^{i} P_{-k \ mod \ n}.pc(F.eopC(-k)) - P_{-k \ mod \ n}.pc(F.sopC(-k)) \leq co)$

$\quad first\ \Sigma_{k=1}^{-frco} P_{-k \ mod \ n}.pc(F.eopC(-k)) - P_{-k \ mod \ n}.pc(F.sopC(-k)) \leq co)$

*This way we get*

$\quad frameCoordinate(co)\ (frco, first)$ ■

The 'start of partition' function *sopC* is now defined as follows:

**Definition 3.7.23 (The Function *sopC* for Folded Partitionings)**
*Let* $P$ *be a folded partitioning with frame partitioning* $F$ *and component partitionings* $P_0, \ldots, P_{n-1}$. *Let co be a coordinate of* $P$.

*Let* $(frco, first)\ frameCoordinate(co)$

*Let border* $= F.sopC(frco)$ *be the left boundary of the frame partition containing the component partition with coordinate co.*

*Now we define*

$\quad sopC(co)\ P_{frco \ mod \ n}.sopC(P_{frco \ mod \ n}.pc(border) + (co - first))$ ■

All operations on folded partitionings are straightforward if the coordinates involved remain within a single frame partition. The corresponding operation on the corresponding component partition can be used in this case. If the coordinates, however, cross the border of a frame partition then a special treatment is necessary. We explain this for the $shiftPD(t, m)$ function. All other functions are similar.

The $shiftPD(t, m)$ function for folded partitionings is straightforward if the shifted time point still remains in the same frame partition. In this case it is sufficient to use the $shiftPD$ function of the corresponding component partitioning. If this shift crosses the border of the frame partition then the relative distance to this border is subtracted from $m$, and then $shiftPD$ is called recursively for the border time point and the reduced $m$-value.

**Definition 3.7.24 (`shiftPD` for Folded Partitionings)** *Let* $P$ *be a folded partitioning with frame partitioning* $F$ *and component partitionings* $P_0, \ldots, P_{n-1}$. *Let* $t$ *be a time point and* $m$ *a real number.*

$P.shiftPD(t, m)$ *performs the following steps:*

1. *Let co* $F.pc(t)$ *be the coordinate of the frame partition containing* $t$.
2. *Let* $C\ P_{co \ mod \ n}$ *the component partitioning assigned to the partition containing* $t$.
3. *Let* $s\ C.shiftPD(t, m)$ *be the shifted time point.*
4. *If* $s$ *is still in the same frame partition as* $t$, *return* $s$.
5. *If* $s$ *is not the in the same frame partition, let* $m'$ *be the relative distance between* $t$ *and the end* $t'$ *(if* $m > 0$*) or the start* $t'$ *(if* $m < 0$*) of the frame partition containing* $t$.
6. *Subtract* $m'$ *from* $m$ *giving* $m''$, *and call* $P.shiftPD(t', m'')$ *recursively.*

■

## 3.8 Summary

The basic ideas of the PartLib library for representing periodic temporal notions are explained in this chapter. We use partitionings of the real numbers as basic mathematical structures. The partitionings can be labelled with symbolic names. The labels can be used for various purposes, in particular for defining *granules*, i.e. clusters of partitions which belong together semantically.

The approach proposed in this chapter is a mix of algorithmic and symbolic specifications. The basic time units are realized as algorithmic partitionings where the details of the calendar system is hard-coded in the correction functions. All other periodic temporal notions are specified symbolically as regular, duration or folded partitionings. This seems to be a good compromise between the efficiency of compiled code for the difficult parts of calendar systems, and the flexibility of symbolic specifications for application specific parts.

The PartLib library is an open source C++ system. The details of its interface are explained in the appendix.

## 3.9 Appendix: The PartLib Interface

PartLib is an object oriented C++ implementation of the ideas presented in this chapter. The interface to this library is presented here in a slightly abstracted way. We omit the technicalities of objects, pointers, references, const declarations etc. and just show the logical aspects of the interface. The technical details can be taken from the corresponding header files.

Nevertheless, certain technicalities are still important. One of them are the datatypes. We need 'int' (integers) and 'string' (strings) as basic datatypes. Floating point numbers are indicated by the type 'FLT'. This must be either 'float', 'double' or 'long double'. 'float' has only a precision of 6 digits, which is in general too small compared to a date like 2004/12/4/12/43/22, which has 13 digits. Therefore 'double' is recommended. Furthermore we use 'Rt' (for Reference time) and 'Co' (for Coordinate). They must both be integer datatypes. At least 64 bit integers are recommended. Bignumbers might be even better. Co and Rt can be the same or different. Since Co and Rt can be the same datatypes, PartLib cannot use overloaded functions in most cases. Therefore a number of function names have suffixes 'Co' and 'Rt'.

All the classes which are defined in PartLib can of course also be used as datatypes. These are in particular 'Label', 'Labelling', 'Duration', 'DateFormat', 'DateData', 'Partitioning', and the subclasses of 'Partitioning'.

The notation for the methods (functions) used in this appendix is

ResultType methodName(datatype$_1$ parameter$_1$,...)

'ResultType' is the datatype of the resulting value. If 'ResultType' is omitted then no result is returned by the method. 'datatype$_1$ parameter$_1$' etc. declares the datatypes of the parameters. 'datatype$_n$ parameter$_n$ = value' is also possible. It means that the default value for this parameter is 'value'.

NULL is the 'null pointer' which indicates that there is no object.

### 3.9.1 Repetition Patterns

Various classes defined below have constructor functions which accept part of their key parameters as strings. For example, the Labellings class has a constructor function which accepts the sequence of labels as a string of label names. These constructor functions make it easier to process user specified concepts by just passing the strings from the user input or from a specification file to the PartLib functions. Moreover, it allows the user to use shortcuts for repeating patterns of the same information.

As an example, where the usefulness of this becomes obvious, suppose we want to specify a bus timetable as a regular or duration partitioning. The concrete timetable is completely specified by listing the dates for Monday, declaring that this list is repeated 5 times (for Monday until Friday), then listing the dates for Saturday, and declaring that this second list is repeated twice. A compact notation for this would be something like '5*(list for Monday), 2*(list for Saturday)'. This compact notation can then be expanded by repeating the list for Monday five times, and concatenating it with two copies of the list for Saturday.

To enable this shortcut notation, we define a the formal language of *repetition patterns*.

**Definition 3.9.1 (Repetition Patterns)** *Let $B$ be a list of* base strings. *The base strings could for example be the string representation of integers, or the label names, or duration specifications. A* repetition pattern *$R$ over $B$ is the following formal language:*

1. *each element of $B$ is a repetition pattern;*
2. *an expression $n * b$, where $n$ is a non-negative integer and $b$ is an element of $B$, is a repetition pattern;*
3. *a comma separated sequence of repetition patterns is repetition pattern;*
4. *an expression $n * (r)$, where $n$ is a non-negative integer and $r$ is a repetition pattern, is a repetition pattern.*

∎

A repetition pattern can be expanded to a sequence of strings by repeating for $n * b$ the element $b$ $n$ times, and by repeating for expressions $n * (r)$ the expanded element $r$ $n$ times.

**Example 3.9.2 (for Repetition Pattern)** *If $B$ is a set of label names 'l1, l2, l3, l4, l5, gap' then '2*(l1, 2*l2, gap)' is a repetition pattern which expands to 'l1, l2, l2, gap, l1, l2, l2, gap'. '2*(l1, 2*(l2, gap))' expands to 'l1, l2, gap, l2, gap, l1, l2, gap, l2, gap'.* ∎

### 3.9.2 The Label Class

A label is essentially a name (string). The name, however, depends on the *current realm*, which is also a string. A label can have a different name in each realm. The Label class manages the set of realms and the mapping of label names to realms. There is always a 'current realm', such that the name() function (see below) can return a name. By changing the realm one can change the names of all labels. The default realm is 'en'.

#### Constructor Function

`Label(string name)`
: Constructs a new label with the given name. An error is thrown if the name is the empty string.

### Class Methods

`setRealm(string realm)`
: declares a new realm and makes it the current realm. setRealm() throws an error if the realm is the empty string.

`string getCurrentRealm()`
: returns the current realm.

`string getAllRealms()`
: returns all realms as a comma separated list (unordered).

`vector<string> getAllRealmsAsVector()`
: returns all realms as a vector (unordered).

### Public Instance Methods

`string name()`
: returns the name of the label in the current realm. name() throws an error if for the current realm no name is defined for the label.

`Label getLabel(string name)`
: If a label with this name (in the current realm) exists already then this label is returned, otherwise a new label with this name is constructed and returned. getLabel throws an error if the name is the empty string.

`addName(string name, string realm)`
: defines for the given realm a new name for the label. If the realm does not yet exist, it is constructed. addName throws an error if the name or realm are empty strings, or if the label is the gap label. 'gap' labels cannot be renamed.

`bool isGap()`
: returns true if the label is the gap label.

`bool notGap()`
: returns true if the label is not the gap label.

`display()`
: prints all names in all realms to stdout (for debugging purposes).

## 3.9.3 The Labelling Class

A labelling is just a vector of labels. The labelling class has no internal states. It therefore just manages individual labellings.

### Constructor Functions

`Labelling(vector<Label> Labels)`
: constructs a new labelling from the given labels. (Def. 3.2.7)

`Labelling(string Labels)`
: 'Labels' is a repetition pattern over the set of label names (Def. 3.9.1), which expands to a list of label names. If labels with these names do not yet exist, they are constructed. Different occurrences of the same name are mapped to the same label. The name 'gap' is mapped to the unique gap label. A new labelling is constructed with these labels. (Def. 3.2.7)

## Public Instance Variables

`vector<Label> Labels`
: contains the labels

`int size`
: number of labels in the labelling

`int nGaps`
: number of gaps in the labelling

`bool longGranules`
: true if the labelling has granules consisting of more than one partition.

`bool longGranulesOrGaps`
: true if the labelling has long granules or gaps.

## Public Instance Methods

`bool hasGaps()`
: returns true if there are gaps in the labelling

`bool validLabel(string name)`
: returns true if a label with this name exists in the labelling.

`Label label(Co coordinate)`
: returns the label for the given coordinate

`bool isGap(Co coordinate)`
: returns true if the label for the given coordinate is the gap label.

`bool notGap(Co coordinate)`
: returns true if the label for the given coordinate is not the gap label.

`Co nextNonGap(Co coordinate)`
: If the label at the given coordinate is not the gap label then 'coordinate' is returned. If it is the gap label then the coordinate of the next non-gap label is returned.

`Co previousNonGap(Co coordinate)`
: If the label at the given coordinate is not the gap label then 'coordinate' is returned. If it is the gap label then the coordinate of the previous non-gap label is returned.

`bool withinGranule(Co coordinate)`
: returns true if the label at the given coordinate is within a granule.

`Co nextGranule(Co coordinate)`
: returns the start coordinate of the next granule. The label at 'coordinate' may be within a granule or between granules.

`Co previousGranule(Co coordinate)`
: returns the end coordinate of the previous granule. The label at 'coordinate' may be within a granule or between granules.

`Co startOfGranule(Co coordinate)`
: If the label at 'coordinate' is within a granule then the start coordinate of this granule is returned. If the label is between two granules then the start coordinate of the next granule is returned.

`Co endOfGranule(Co coordinate)`
: If the label at 'coordinate' is within a granule then the end coordinate of this granule is returned. If the label is between two granules then the end coordinate of the previous granule is returned.

`Co lastCoordinate(Co coordinate)`
: returns the coordinate of the last non-gap label of the copy of the labelling at the given coordinate.

`Co lastCoordinate(Co coordinate)`
: returns the coordinate of the first non-gap label of the copy of the labelling at the given coordinate.

`int numberOfGaps(Co c1, Co c2)`
: returns the number of gap labels between coordinate c1 and c2 (both inclusive).

`granule(Co coordinate, Co c1, Co c2)`
: binds c1 to startOfGranule(coordinate) and c2 to endOfGranule(coordinate).

`Co nextGranuleWithLabel(Co coordinate, int n, string name)`
: If $n = 0$ then the given coordinate is just returned. If $n > 0$ then the start coordinate of the granule with the $n$th next occurrence of a label with the given name is returned. If $n < 0$ then the start coordinate of the granule with the $n$th previous occurrence of a label with the given name is returned.

`display()`
: prints the labelling in all realms to stdout (for debugging purposes.)

### 3.9.4 The Duration Class

A duration is a list of pairs (number, partitioning), together with a flag 'asGranule'. If this flag is true and there are labellings attached to the partitionings, then certain functions, in particular the shift function, takes the granules as basic units, and not the partitions.

#### Constructor Functions

`Duration(vector<pair<FLT,Partitioning> durations, bool asGranule=false)`
: constructs a new duration.

```
Duration(vector<FLT> shifts, vector<Partitioning> partitionings,
         bool asGranule=false)
```
: constructs a new duration by pairing the shifts and partitionings together.

`Duration(string duration, bool asGranule=false)`
: constructs a new duration from a string representation '⟨number⟩ ⟨partitioning_name⟩, ...', for example, '3.5 month, -1 week, 2 hour'

#### Public Instance Methods

`Duration invert()`
: constructs a new duration by inverting the given one

`append(Duration D)`
: appends the duration D to the given one.

| | | | |
|---|---|---|---|
| Example: | '3 week, 2 day'.append('1 hour, 3 minute') | $\rightarrow$ | '3 week, 2 day, 1 hour, 3 minute' |
| or: | '3 week, 2 day'.append('3 day, 3 minute') | $\rightarrow$ | '3 week, 5 day, 3 minute' |
| but: | '3 week, 2 day'.append('1 week') | $\rightarrow$ | '3 week, 2 day, 1 week' |
| and: | '1.5 week'.append('1.5 week') | $\rightarrow$ | '1.5 week, 1.5 week' |

`Rt shiftOnce(Rt time, Partitioning P=NULL)`
: shifts the time point 'time' by the given duration. If 'asGranule' is true then the time point is shifted by interpreting the partitionings as granules (see 'shiftAsGranule' below), otherwise they are interpreted as partitions. The underlying shift function is either 'shiftPartitiongsbyDate' or shiftGranules, if 'asGranules' = true.

`Rt shift(Rt time, FLT m=1.0, Partitioning P=NULL)`
: shifts the time point 'time' by $m$ times the given duration. $m$ can be any positive of negative FLT number.

`vector<pair<Rt,Rt> > sections(Rt from, Rt to)`
: splits the section [from,to[ of the time axis into subsections $[(t_0, d_0), (t_1, d_1), ...]$, such that the duration causes a shift of $d_i$ basic time units between the time points $t_i$ and $t_{i+1}$. If the time shift is always the same value $d_0$ then only (from,$d_0$) is returned.

#### Relations

`bool operator<(Duration D1, Duration D2)`
: This is a <-operator which compares two durations. It yields true if a shift by D1 is always shorter than a shift by D2. For example, '1 month, 1 day' < '4 week'. So far, PartLib contains only an approximative implementation of this relation: A finite number of time points is generated, and the shifts from these time points are compared. See also Section 3.2.4.

### 3.9.5 Class DateFormat

A date format is a list of partitions and a name. The name can be used to retrieve a previously created date format from the DateFormat class.

#### Constructor Functions

`DateFormat(string name, vector<Partitioning> partitionings)`
: constructs a date format with the given name and partitionings.

`DateFormat(string name, string pnames)`
: constructs a date format with the given name from the comma separated names of the partitionings.

#### Class Methods

`DateFormat getDateFormat(string name)`
: returns a previously generated date format with this name. If the name is unknown then NULL is returned.

**Public Instance Methods**

`DateData date(Rt time, clear = true)`
: turns a time point into dates (see below) of the given date format. If clear = true then trailing zeros are deleted, otherwise the date is exactly as long as the date format. (Def. 3.3.2)

### 3.9.6 Class DateData

A DateData object is essentially a date format and a sequence of integers, the dates.

**Constructor Functions**

`DateData(DateFormat dF, vector<Co> data)`
: constructs a DateData object for the given date format from the data.

`DateData(DateFormat dF, string dD, string separator = "/")`
: constructs a DateData object for the given date format from a string representation of the data. The data must be a sequence of positive or negative integers, separated by the given separator. An error is thrown if the string cannot be parsed properly.

**Public Instance Methods**

`Rt TimePoint()`
: turns the DateData object into the corresponding time point. (Def. 3.3.3)

`append(vector<Co> date, Co extra)`
: appends 'date' at the end of the given date and adds 'extra' to date[0].

### 3.9.7 The Partitioning Class

The 'Partitioning' class consists of the abstract class 'Partitioning' and a few subclasses. The class diagram is:

```
Partioning
 |
 |---- algorithmicPartitioning
 |---- regularPartitioning
 |---- durationPartitioning
 |       └---- datePartitioning
 └---- foldedPartitioning
```

The partitionings are parameterized with an instance of a class 'CalContext'. This class is not defined in PartLib. It must provide the following methods:

- *intresolution*(): which returns a positive integer. 1 means, the basic time unit are seconds. 1000 means, the basic time unit are milliseconds.
- *inttimezone*() returns the timezone as an integer.

- *leapSeconds*() returns a LeapSeconds object, which has the two methods *globalCorrection*(*grt*) (*lsG* in Def. 3.6.1) and *localCorrection*(*lrt*) (*lsL* in Def. 3.6.1).

### The Abstract Class 'Partitioning'

This class is essentially the interface to the PartLib library. Except for the constructor functions of the subclasses, one should use only the public methods from this class.

### General Methods

`name`
: is the public instance variable which holds the name of the partitioning.

`static Partitioning getPartitioning(string name)`
: returns the partitioning with the given name, or NULL if this name is unknown.

`static vector<Partitioning> allPartitionings();`
: returns a vector of all partitionings.

### Coordinate Computations

`Rt startOfPartitionRt(Rt time)`
: maps the given time point to the start time of the partition containing this point. (Def. 3.2.3)

`Rt endOfPartitionRt(Rt time)`
: maps the given time point to the end time of the partition containing this point. (Def. 3.2.3)

`Rt startOfPartitionCo(Co coordinate)`
: maps the given coordinate to the start time of the partition with this coordinate (cf. *sopC*). (Def. 3.2.3)

`Rt endOfPartitionCo(Co coordinate)`
: maps the given coordinate to the end time of the partition with this coordinate (cf. *eopC*). (Def. 3.2.3)

`Co partitionCoordinate(Rt time)`
: maps a time point to the coordinate of the partition containing this point (cf. *pc*). (Def. 3.2.3)

### Labels and Granules

`setLabelling(Labelling L)`
: attaches the labelling L to the given partitioning. This method throws an error for folded partitionings.

`bool validLabel(string label)`
: returns true if the label occurs in the labelling.

`Labelling labellingCo(Co coordinate)`
: returns the labelling which is valid for the given coordinate and NULL if there is none. This is non-trivial only for folded partitionings.

`Labelling labellingRt(Rt time)`
: returns the labelling which is valid for the given time, and NULL if there is none. This is non-trivial only for folded partitionings.

`labelCo(Co coordinate)`
: returns the label which is attached to the given coordinate, and NULL if there is none.

`labelRt(Rt time)`
: returns the label which is attached to the partition containing this time, and NULL if there is none.

`bool isGap(Co coordinate)`
: returns true if the label at this gap is the gap label.

`bool notGap(Co coordinate)`
: returns true if the label at this gap is not the gap label.

`bool hasGaps(Co coordinate)`
: returns true if the labelling which is valid for this coordinate has gaps. The coordinate plays only a role for folded partitionings.

`bool withinGranuleCo(Co coordinate)`
: true if the coordinate is within a granule.

`bool withinGranulRt(Rt time)`
: true if the time point is within a granule.

`bool closestGranuleCo(Co coordinate, Co c1, Co c2)`
: binds c1 and c2 to the start and end coordinates of the granule which is closest to 'coordinate'. The result value is true if the coordinate is within a granule. (Def. 3.2.10)

`bool closestGranuleRt(Rt time, Co c1, Co c2)`
: binds c1 and c2 to the start and end coordinates of the granule which is closest to the given time point. The result value is true if the time point is within a granule. (Def. 3.2.10)

### Local Length of Partitions and Granules

`Rt lengthOfPartitionCo(Co coordinate)`
: returns the length of the partition with the given coordinate.

`Rt lengthOfPartitionRt(Rt time)`
: returns the length of the partition containing the given time point.

`FLT lengthInPartitions(Rt t1, Rt t2)`
: returns the length of the interval [t1,t2[ in terms of the length of the partitions of the given partitioning. (Def. 3.2.5)

`FLT lengthInGranules(Rt t1, Rt t2)`
: returns the length of the interval [t1,t2[ in terms of the length of the granules of the given partitioning. (Def. 3.2.11)

### Global Length of Partitions and Granules

`setShortestPartition(Rt length)`
: overrides the randomized computation of partition length by just declaring the shortest partition length.

`setLongestPartition(Rt length)`
: overrides the randomized computation of partition length by just declaring the longest partition length.

`Rt shortestPartition()`
: returns the length of the shortest partition.

`Rt longestPartition()`
: returns the length of the longest partition.

`Rt shortestGranule()`
: returns the length of the shortest granule.

`Rt longestGranule()`
: returns the length of the longest granule.

`bool hasShorterPartitionsThan(Partitioning Q)`
: returns true if the given partitioning has shorter partitions than partitioning Q. (Def. 3.2.12)

`bool hasShorterGranulesThan(Partitioning Q)`
: returns true if the given partitioning has shorter granules than partitioning Q. (Def. 3.2.12)

### Shift of Time Points

`Rt shiftPartitionsbyLength(Rt time, FLT m, DateFormat dF = NULL, Partitioning P = NULL`
: shifts the time point by the length of $m$ partitions. If a date format $dF$ and a partitioning $P$ is given then the shifted time point is moved again such that the dates of the shifted time and the original time are kept the same from the partitioning $P$ onwards. For example, if the date format is year/month/day/hour/minute/second, and P = hour, then the shifted time has the same hour/minute/second date as the original time. (Def. 3.4.2)

`shiftPartitionsbyLength(Rt time, FLT m, string dF, string P)`
: This method is almost identical to the above method. The only difference is that the date format and the partitioning are given by their names, and not by the pointers. (Def. 3.4.2)

`Rt shiftPartitionsbyDate(Rt time, FLT m, Partitioning P = NULL`
: shifts the time point by $m$ partitions using the dates for the shift. If a partitioning $P$ is given then the shifted time point is moved again such that the dates of the shifted time and the original time are kept the same from the partitioning $P$ onwards. (Def. 3.7.5)

`Rt shiftPartitionsbyDate(Rt time, FLT m, string P`
: This method is almost identical to the above method. The only difference is that the partitioning is given by its names, and not by the pointer. (Def. 3.7.5)

`Rt shiftGranules(Rt time, FLT m, Partitioning P = NULL)`
: shifts the time point by $m$ granules. If a partitioning $P$ is given then the shifted time point is moved again such that the dates of the shifted time and the original time are kept the same from the partitioning $P$ onwards, where $P$ is a partitioning with a corresponding date format. This method uses internally the ‘shiftPartitionsbyDate’ method. shiftGranules throws an error if the time point is between two granules. (Sect. 3.4.3)

`Rt shiftGranules(Rt time, FLT m, string P)`

  This method is almost identical to the above method. The only difference is that the partitioning is given by its names, and not by the pointer. (Sect. 3.4.3)

`Co nextGranuleWithLabel(Co coordinate, int n, string label)`

  If $n = 0$ then 'coordinate' is returned. If $n > 0$ then the coordinate of the start of the $n$th next granule with the given label name is returned. If $n < 0$ then the coordinate of the end of the $n$th previous granule with the given label name is returned.

### Structural Relations

`bool includesPartitions(Partitioning Q)`

  returns true if each partition of the given partition is a subset of a partition of Q. (Def. 3.2.12)

`bool includesGranules(Partitioning Q)`

  returns true if each granule of the given partition is a subset of a granule of Q. (Def. 3.2.12)

`bool subsetGranule(Rt time, Co coP, Partitioning Q, Co coQ)`

  returns true if the P-granule at coordinate coP is a subset of the Q-granule at coordinate coQ. (Def. 3.2.12)

`bool operator<(Partitioning P, Partitioning Q)`

  $P < Q$ is just short for P.hasShorterPartitionsThan(Q). (Def. 3.2.12)

### Boundaries

`bool leftBounded`

  is a public instance variable, which is true when a left boundary is defined.

`Rt leftBoundary`

  is an instance variable which holds the left boundary.

`bool rightBounded`

  is a public instance variable, which is true when a right boundary is defined.

`Rt rightBoundary`

  is an instance variable which holds the right boundary.

`void setLeftBoundary(Rt time)`

  sets the left boundary.

`void setRightBoundary(Rt time)`

  sets the right boundary.

### Subclasses of the Partitioning class

Only the constructor functions for the subclasses are in the interface. All other methods are to be called via the Partitioning class interface.

`algorithmicPartitioning(string name, int avl, Co2Rt cf, Rt offset, CalContext C, int repetitions, Labelling L=NULL)`

  constructs an arithmetic partitioning. 'name' is the name of the partitioning, 'avl' is the average length of the partitions. 'cf' is the correction function. 'offset' is the offset of the partition with coordinate 0. 'C' is the context object, 'repetitions' is the parameter for the randomized checks, and 'L' is a labelling. (Def. 3.7.1) `regularPartitioning(string name, Rt anchor, vector<FLT> sizes, Partitioning U, CalContext C, bool asGranule, Labelling L=NULL)`

constructs a regular partitioning.'name' is the name of the partitioning. 'anchor' is the anchor time. 'sizes' specifies the lengths of the partitions in terms of the partitioning 'U'. 'C' is the context object. 'asGranule = true' causes the sizes to be interpreted as granule length. 'L' is a labelling. (Def. 3.7.10)

`regularPartitioning(string name, Rt anchor, string sizes, string U, CalContext C, bool asGranule, Labelling L=NULL)`

is almost the same as the above constructor function. The difference is that the sizes are given as repetition pattern over numbers (Def. 3.9.1) and the partitioning 'U' is given by its name. This function throws an error if 'U' is unknown or if 'sizes' cannot be parsed. (Def. 3.7.10)

`durationPartitioning(string name, Rt anchor, vector<Duration> durations, CalContext C, Labelling L=NULL)`

constructs a duration partitioning. 'name' is the name of the partitioning. 'anchor' is the anchor time. 'durations' specifies the lengths of the partitions as durations 'C' is the context object. 'L' is a labelling. (Def. 3.7.7)

`durationPartitioning(string name, Rt anchor, string durations, CalContext C, Labelling L=NULL)`

constructs a duration partitioning. 'name' is the name of the partitioning. 'anchor' is the anchor time. 'durations' is a repetition pattern over duration specifications in quotes (see the constructor function 'Duration'). An example is '2*("1 month, 1 week", "2 day, -1 hour")'. The pattern expands to a sequence of durations. They specify the lengths of the partitions as durations 'C' is the context object. 'L' is a labelling. (Def. 3.7.7)

`datePartitioning(string name, DateFormat dateFormat, vector<vector<Co>> dates, CalContext C, Labelling L=NULL)`

constructs a date partitioning. 'name' is the name of the partitioning. 'dateFormat' is the date format for the dates. 'dates' is the vector of dates. 'C' is the context object. 'L' is a labelling. (Def. 3.7.16)

`datePartitioning(string name, DateFormat dateFormat, string dates, CalContext C, Labelling L=NULL)`

This constructor function is almost like the above function. The difference is that the dates are given as ' '-separated dates (which are '/'-separated integers). (Def. 3.7.16)

`foldedPartitioning(string name, Partitioning framePartitioning, vector<Partitioning> componentPartitionings, bool constant, CalContext C)`

constructs a folded partitioning. 'name' is the name of the partitioning. 'framePartitioning' is the frame partitioning and 'componentPartitionings' is the vector of component partitionings. The 'constant' flag indicates whether this is a constant partitioning. This is not checked! If a folded partitioning is declared constant, but is not, the results of all computations are unpredictable. (Def. 3.7.17)

`foldedPartitioning(string name, Partitioning framePartitioning, string componentPartitionings, bool constant, CalContext C)`

This constructor function is almost like the above function. The difference is that the component partitionings are given as repetition pattern over the partitioning names (Def. 3.9.1). This function throws an error if partitioning names are unknown. (Def. 3.7.17)

Chapter 4

# Calendrical Calculations with Time Partitionings and Fuzzy Time Intervals

**Hans Jürgen Ohlbach**

**Abstract**

This chapter presents a piece in a big mosaic which consists of formalisms and software packages for representing and reasoning with everyday temporal notions. The kernel of the mosaic consists of several layers. At the bottom layer there are a number of basic datatypes for elementary temporal notions. These are time points, crisp and fuzzy time intervals and partitionings for representing periodical temporal notions like years, months, semesters etc. Partitionings can be arranged to form 'durations' (e.g. '2 semester and 1 month'). Each formalism in the bottom layer comes with its own functions and relations.

The second layer is presented in this chapter. It contains a number of basic functions which use time points, intervals, partitionings and durations simultaneously. The functions are introduced and motivated with temporal expressions in natural language.

The third layer, which is not presented in this chapter, uses the functions and relations of the lower layers as building blocks in a specification language for specifying complex temporal notions.

The whole mosaic contains a number of other formalisms, in particular a representation of calendar systems, and various databases with information about temporal notions.

## 4.1 Motivation and Introduction

The phenomenon of *time* has many different facets which are investigated by different communities. Physicists investigate the flow of time and its relation to physical objects and events. Temporal logicians develop abstract models of time where only the aspects of time are formalized which are sufficient to model the behaviour of computer programs and similar processes. Linguists develop models of time which can be used as semantics of temporal expressions in natural language. More and more information about facts and events in the real world is stored in computers, and many of them are annotated with temporal information. Therefore it became necessary to develop computer models of the use of time on our planet, which are sophisticated enough to allow the kind of computation and reasoning that humans can do. Examples are 'calendrical calculations' [16], i.e. formal encodings of calendar systems for mapping dates between different calendar systems. Other models of time have been developed in the temporal database community [10], mainly for dealing with temporal information in databases. This work is becoming more important now with the emergence of the Semantic Web [7]. Informal, semi-formal and formal temporal notions occur frequently in XML documents, and need be 'understood' by XML query and transformation mechanisms.

The formalisms developed so far approximate the real use of time on our planet to a certain degree, but still ignore important aspects. In the WebCal project [12] we aim at a very detailed modelling of the temporal notions which can occur in semi-structured data. The WebCal system consists of a kernel and several modules around the kernel. The kernel itself consists of several layers. At the bottom layer there are a number of basic datatypes for elementary temporal notions. These are time points, crisp and fuzzy time intervals [40, 41], and partitionings for representing periodical temporal notions like years, months, semesters etc. [43, 42]. The partitionings can be specified algorithmically or algebraically. The algorithmic specifications allows one to encode phenomena like leap seconds, daylight savings time regulations, the Easter date, which depends on the moon cycle etc. Partitionings can be arranged to form 'durations' (e.g. '2 semester and 1 month'). Each formalism in the bottom layer comes with its own functions and relations.

The second layer, the *mixed function layer*, is presented in this chapter. It contains a number of basic functions which use time points, intervals, partitionings and durations simultaneously. The functions are introduced and motivated with temporal expressions in natural language.

The third layer, which is not yet worked out in detail, uses the functions and relations of the lower layers as building blocks in a specification language for specifying complex temporal notions. A first version of this language has been presented in [37, 38].

Since the mixed function layer uses the formalisms from the first layer, we briefly introduce this layer and then define the mixed functions.

## 4.2 Time Points and Time Intervals

The flow of time underlying most calendar systems corresponds to a time axis which is isomorphic to the real numbers $\mathbb{R}$. Therefore we take as time points just real numbers. Since the most precise clocks developed so far, atomic clocks, measure the time in discrete units, it would be sufficient to resrict the representation of time points to integers, but this is not important for the purposes of this chapter. It becomes an issue for concrete implementations.

The next important datatype is that of time intervals. Time intervals can be crisp or fuzzy. With fuzzy intervals one can encode notions lake 'around noon' or 'late night' etc. This is more general and more flexible than crisp intervals. Therefore the WebCal system uses fuzzy intervals as basic interval datatype.

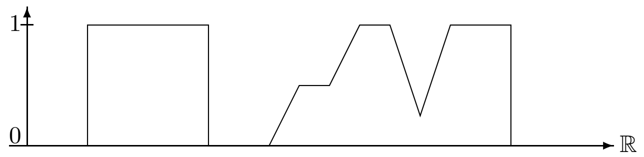
Fuzzy Intervals are usually defined through their membership functions [53, 17]. A membership function maps a base set to a real number between 0 and 1. The base set for fuzzy time intervals is a linear time axis, isomorphic to the real numbers.

**Definition 4.2.1 (Fuzzy Time Intervals)** *A* fuzzy membership function *is a total function*

*$f : \mathbb{R} \mapsto \mathbb{F}$ which does not need to be continuous, but it must be integratable. The* fuzzy interval *$i_f$ that corresponds to a fuzzy membership function $f$ is $i_f$ $\{(x, y) \subseteq \mathbb{R} \times \mathbb{F} \mid y \leq f(x)\}$. Given a fuzzy interval $i$ we usually write $i(x)$ to indicate the corresponding membership function.*

*A fuzzy time interval may consist of several subintervals or* components*. Let $Comp(i)$ the the components of $i$.* ■

This definition comprises single or multiple crisp or fuzzy intervals like these:



*Crisp and Fuzzy Intervals*

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

Fuzzy time intervals may be quite complex structures with many different characteristic features. The simplest ones are *core* and *support*. The core $C(i)$ is the part of the interval $i$ where the fuzzy value is 1, and the support $S(i)$ is the subset of $\mathbb{R}$ where the fuzzy value of $i$ is non-zero. In addition one can define the *kernel* $K(i)$ as the part of the interval $i$ where the fuzzy value is *not* constant ad infinitum, i.e. the kernel is the smallest convex interval in $\mathbb{R}$ such that $i(x)$ is constant for $x$ outside $K(i)$. 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.

Fuzzy time intervals can be measured in various ways. Besides the size $|i|$ $\int i(x)\, dx$, one can locate the position of the core, support and kernel. $i^{fO}$ and $i^{lO}$ are the first and last time coordinates of $O$, where $O$ is either $C$ (core) or $S$ (support) or $K$ (kernel). One can also measure the maximal fuzzy value $i\hat{}$. This should, but need not be 1. Let $i^{fm}$ be the time coordinate of the first point with $i(i^{fm}) = i\hat{}$ and let $i^{lm}$ be the last such point.

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

**Definition 4.2.2 (Envelope Polygon)** *The* envelope polygon *$I$ of a fuzzy time interval is a finite sequence of points $p_0, \ldots, p_n$. Each point $p$ is a tuple $(p.x, p.y)$ consisting of the time coordinate $p.x$ and the fuzzy value coordinate $p.y$.*

*$p_i.x \leq p_{i+1}.x$ must hold for all $i$.*

*We usually identify the envelope polygon with the fuzzy set itself.* ■

The details of the fuzzy intervals and the operations used in the WebCal system are presented in [40, 41]. The only aspect of fuzzy time intervals which is of some importance for this chapter are the set operations $\cup$ (union), $\cap$ (intersection) and $\setminus$ (set difference) for fuzzy intervals. There is no unique definition of these operations for fuzzy intervals. Therefore one has to state, which definition is being used. A definition of intersection which takes the minimum of the membership functions, and a corresponding definition of union which takes the maximum of the membership functions is very natural, but not the only choice. In this chapter we can leave it open, which version is to be used. In the implementation one can choose a particular one, or one can leave it as an extra parameter, and the application system has to make the choice.

## 4.3 Partitionings

The WebCal system uses the concept of *partitionings* of the real numbers to model periodical temporal notions. In particular, the basic time units years, months etc. are realized as partitionings. 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 natural numbers (their *coordinates*). For example, we could have the following enumeration

| | | | | | |
|---|---|---|---|---|---|
| ... | $[-100\ 0[$ | $[0\ 100[$ | $[100\ 101[$ | $[101\ 500[$ | ... |
| ... | $-1$ | $0$ | $1$ | $2$ | ... |

We use *labelled partitionings.* The labels are names for the partitions. For example, the labels for the 'day' partitioning can be 'Monday', 'Tuesday' etc.

The formal definition for labelled partitionings of $\mathbb{R}$ which is used in this chapter is:

**Definition 4.3.1 (Labelled Partitionings)** *A partitioning $P$ of the real numbers $\mathbb{R}$ is a sequence $\ldots [t_{-1}, t_0[, [t_0, t_1[, [t_1, t_2[, \ldots$*
*of half open intervals in $\mathbb{R}$ with integer boundaries, such that either 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*).*

*For a time point $t$ and a partitioning $P$ let $t^P$ be the $P$-partition containing $t$.*

*A* coordinate mapping *of a partitioning $P$ is a bijective mapping between the intervals in $P$ and a subset of the integers. Since we always use one single coordinate mapping for a partitioning $P$, we can just use $P$ itself to indicate the mapping. Therefore let $p^P$ be the* coordinate *of the partition $p$ in $P$.*

*For a coordinate $i$ let $i^P$ be the partition which corresponds to $i$.*

*For a time $t$ let $t^{PP}$ $(t^P)^P$ be the coordinate of the $P$-partition containing $t$.*

*A* Labelling *$L$ is a finite sequence of strings $l_0, \ldots, l_{n-1}$.*

*A labelling $L = l_0, \ldots, l_{n-1}$ is turned into a* labelling function *$L_P(p)$ for a partitioning $P$ as follows:*

$$L_P(p) \begin{cases} l_{p^P \ mod \ n} & \text{if } p \text{ is finite} \\ \text{'startPartition'} & \text{if } p = [-\infty, t[ \\ \text{'endPartition'} & \text{if } p = [t, +\infty[ \end{cases}$$

*where $p$ is a partition in $P$.* ∎

An important operation which involves partitionings is the '*shift*' function. $shift(t, 1, month)$, for example, shifts the time point $t$ by one month. Since the length of the partitions may be different, such a shift function is usually *not continuous.* For example, if the time point $t$ is in January, a shift of 1 month may mean a shift of 31 days, whereas if $t$ is in February, a shift of 1 month may mean a shift of 28 days. The details of the *shift* functions for time points are not important for this chapter. We must, however, keep in mind that the *shift* function is in general not continuous.

The *shift* function can have an extra Boolean parameter *SG* (for Skip Gaps). If it is true then all partitions labelled 'gap' are skipped when the actual shift is computed. All functions defined below can have this extra parameter when they use the $shift$ function. The formalism of labelled partitionings, together with various algorithmic and symbolic specification mechanisms for labelled partitionings, is presented in [43].

## 4.4 Durations

The partitionings are the mathematical model of periodic time units, such as years, months etc. This offers the possibility to define *durations*. A duration may, for example, be '3 months and 2 weeks'. Months and weeks are represented as partitionings, and 3 and 2 denote the number of partitions in these partitionings. The numbers need not be integers, but can be arbitrary real numbers.

A duration can be interpreted as the length of an interval. In this case the numbers should not be negative. A duration, however, can also be interpreted as a time shift. In this interpretation negative numbers make perfectly sense. $d = (-2\ week), (3\ month)$, for example, denotes a backward shift of 2 weeks followed by a forward shift of 3 months.

**Definition 4.4.1 (Duration)** *A* duration $d = (d_0\ P_0), \ldots, (d_k\ P_k)$ *is a list of pairs where the* $d_i$ *are real numbers and the* $P_i$ *are partitionings.*

*If a duration is interpreted as a shift of a time point, it may be necessary to turn the shift around, in the backwards direction. Therefore the inverse of a duration is defined:*
$-d\ (-d_k\ P_k), \ldots, (-d_0\ P_0)$ ■

For example, if $d = (3\ month), (2\ week)$ then $-d = (-2\ week), (-3\ month)$.

**Definition 4.4.2 (`shift` for Durations)** *Given a function* $shift(t, m, P)$*, which shifts a time point* $t$ *by* $m$ *partitions of the partitioning* $P$*, we define a corresponding* `shift` *function for durations:*

$$shift(t, d)\ \ shift(\ldots shift(shift(t, d_0, p_0), d_1, p_1) \ldots, d_k, p_k)$$

*where* $t$ *is a time point and* $d = (d_0, p_0), \ldots, (d_k, p_k)$ *is a duration.* ■

## 4.5 Operations on Points, Intervals and Partitionings

In this section we introduce a number of operations which involve points, intervals, partitionings and durations simultaneously. Together with the operations for the individual datatypes they form the basic building blocks for a specification language for temporal notions.

Since the intervals are represented by their envelope polygons, there is the choice to define the operations directly on the envelope polygons, or on the abstract notion of interval. In the implementation they must operate on the envelope polygons, but it makes sometimes things clearer if they are defined on the abstract level.

It turned out that even quite simple operations become very complex and need many parameters if all the details of fuzzy intervals and partitionings are taken into account. Therefore it is necessary to introduce the operations informally first, and to motivate and discuss the details before the formal definition is presented.

## 4.5.1 Shift and Extend

The first operation is the 'shift' operation for intervals. Suppose we have an interval of two hours, and want to shift it by one week. The two hours may be the time for a meeting, and the meeting is shifted to next week. This sounds simple, but the details can be really tricky. The problem is that durations (1 month, 1 week etc.) can denote intervals of different length. In January, for example, a duration of 1 month means 31 days. In February it means only 28 days. This causes that the shift function for time points is not continuous. The distance $(shift(t, 1\ month) - t)$ depends on the position of $t$, and this holds for all meaningful definitions of '$shift$'. As an extreme case, take the interval [January 31, February 1[. If we shift the two time points separately, we can get [February 31, March 1[, which is equivalent to [March 3, March 1[. That is definitely not what one wants. In order to avoid that the intervals get distorted, one must choose a time point as the anchor point for the shift, and compute the size of the shift for this anchor point. If we choose January 31 in the above example as anchor point, we get a shift of 31 days, and the result is [March 3, March 5[. If we choose February 1, the result is [March 1, March 3[.

In principle one can choose any time point as anchor point, but this causes another problem. If the interval consists of several subintervals, it might not be desirable to choose one anchor point for all subintervals. Therefore the shift function defined below uses a symbolic representation for the anchor point. This way the shift function allows one to shift the subintervals separately. For example, if the interval consists of one subinterval in January and one in February, and we shift it by 1 month, then the interval in January is shifted by 31 days, and the interval in February is shifted by 28 days.

**Definition 4.5.1 (shift)** *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon. Let $D$ be a duration. 'componentwise' is a Boolean flag. 'anchorpoint' is one of the keywords 'fm' (first maximum), 'fS' (first Support), 'fC' (first Core), 'fK' (first Kernel), 'lm' (last maximum), 'lS' (last Support), 'lC' (last Core), 'lK' (last Kernel).*

*We define the function $shift(I, D, componentwise, anchorpoint)$ first for the case $componentwise = false$:*

$$shift(I, D, false, anchorpoint)\ \ ((p_0.x + d, p_0.y), \ldots, (p_n.x + d, p_n.y))$$

*where $d\ shift(I^{anchorpoint}, D) - I^{anchorpoint}$.*
*In the case $componentwise = true$, the components of the interval are shifted separately.*

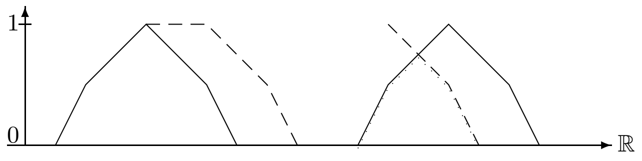
$$shift(I, D, true, anchorpoint)\ \ \bigcup_{J \in Comp(I)} shift(J, D, false, anchorpoint)$$

*The result is undefined if the anchor point is the infinity.* ■

This definition of 'shift' guarantees that finite intervals are not distorted. The gaps between components, however, can be distorted if *componentwise = true* is chosen. It can even happen that the shifted components overlap or they change their ordering.

**extend:** The 'extend' function defined below extends an interval by a given duration. It corresponds to phrases like 'for two more hours'. It can also shrink intervals when the duration is negative. 'extend' is quite similar to the 'shift' function. The difference is that 'extend' shifts only a part of the interval to make it wider or shorter. The choices we have for the 'extend' function are: one can extend the interval at the left or at the right side; one can extend only the outermost subinterval, or all subintervals separately, and one can choose the anchor point for computing the shift.

An extended fuzzy interval has still the same shape as the original one. It has only become longer. It is also possible to shrink a fuzzy interval while preserving its shape more or less. To do this one has to cut it into two pieces, shift one piece and compute the intersection of the first piece with the shifted second piece.



*Extending and Shrinking a Fuzzy Interval*

**Definition 4.5.2 (extend)** *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon. Let $D$ be a duration. 'front' and 'componentwise' are Boolean flags. anchorpoint is one of the keywords 'fm' (first maximum), 'fS' (first Support), 'fC' (first Core), 'fK' (first Kernel), 'lm' (last maximum), 'lS' (last Support), 'lC' (last Core), 'lK' (last Kernel).*
*We define $extend(I, D, front, componentwise, anchorpoint)$ first for the case $componentwise = false$:*

*Let $d$ $shift(I^{anchorpoint}, D) - I^{anchorpoint}$, and let $p_l$ $I^{fm}$ and $p_k$ $I^{lm}$.*

$$extend(I, D, true, false, anchorpoint)$$
$$\begin{cases} ((p_0.x - d, p_0.y), \ldots, (p_l.x - d, p_l.y), p_l \ldots, p_n) & \text{if } d > 0 \\ ((p_0.x - d, p_0.y), \ldots, (p_l.x - d, p_l.y)) \cap (p_l \ldots, p_n) & \text{otherwise.} \end{cases}$$

$$extend(I, D, false, false, anchorpoint)$$
$$\begin{cases} (p_0, \ldots, p_k, (p_k.x + d, p_0.y), \ldots, (p_n.x + d, p_l.y)) & \text{if } d > 0 \\ (p_0, \ldots, p_k) \cap ((p_k.x + d, p_0.y), \ldots, (p_n.x + d, p_l.y)) & \text{otherwise.} \end{cases}$$

*In the case $componentwise = true$, the components of the interval are extended separately.*

$$extend(I, D, front, true, anchorpoint)$$
$$\bigcup_{J \in Comp(I)} extend(J, D, front, false, anchorpoint).$$

*The result is undefined if the anchor point is the infinity.* ■

### 4.5.2 Which

The next function can be used to answer queries like 'which day in the week is now' or 'in which week in the year is the time point $t$'. The parameters are $which(t, P, Q, inclusion, SG)$. $t$ is the time point for which we want to get the information. $P$ and $Q$ are partitionings. For example, $P$ could be the 'week' partitioning and $Q$ could be the 'year' partitioning. *inclusion* is a control parameter for determining what counts as the first $P$ partition in the $Q$ partitioning. The problem can be best illustrated with weeks and years. Since weeks and years are not synchronized, the beginning of a year need not be the beginning of a week. It is therefore a matter of convention what counts as the first week in a year. It could be the first week which is completely contained in the year, it could be the first week which overlaps with the year, or it could be the first week whose lager part is in the year. The latter one is the condition which is commonly being used. Therefore *inclusion* is one of the keywords 'subset', 'overlap' and 'bigger_part_inside'.

The last parameter, $SG$ (for Skip Gaps) controls whether partitions labelled 'gap' are counted or not. If $SG = true$ then they are skipped, otherwise they are counted.

The 'which' function is a partial function. There are various conditions which can cause $which(t, P, Q, inclusion, IG)$ to be undefined. For example, if $SG = true$ and the label of $t^Q$ is 'gap' then the function call is undefined. Another example is when the role of $P$ and $Q$ is reversed and we ask, for example, 'which year in the week is now', with the condition $inclusion = subset$. No meaningful answer is possible in this case.

'which' first locates the $Q$-partition containing $t$. Then it determines the first $P$-partition in this $Q$-partition. Finally it counts the number of $P$-partitions which must be traversed until $t$ is reached.

**Definition 4.5.3 (which)** *Let $t$ be a time point, let $P$ and $Q$ be partitionings. $SG$ is a Boolean flag, and 'inclusion' is one of the keywords 'subset', 'overlap' and 'bigger_part_inside'. $P$ and $Q$ can be labelled.*

$$which(t, P, Q, inclusion, IG)$$
$$\begin{cases} undefined & \text{if } SG = true \text{ and } L_Q(t^Q) = \text{'gap'} \\ undefined & \text{if } n \text{ is undefined} \\ \min_k(t \in (n+k)^P) & \text{if } SG = false \\ \min_k(t \in (n+k)^P) - gaps(n,k) & \text{if } SG = true \text{ and } L((n+k)^P) \neq \text{'gap'} \\ undefined & \text{otherwise} \end{cases}$$

*where $n$ is the coordinate of the first $P$-partition within the $Q$-partition that contains $t$:*

$$n \begin{cases} n' & \text{if } n' \text{ is defined and } (SG = false \text{ or } L_P(n'^P) \neq \text{'gap'}) \\ undefined & \text{otherwise} \end{cases}$$

*The computation of $n'$ depends on the 'inclusion' parameter.*

**Case** *$inclusion = subset$:*

$$n' \begin{cases} (t^Q{}_{[})^{PP} & \text{if } (t^Q{}_{[})^P \subseteq t^Q \\ (t^Q{}_{[})^{PP} + 1 & \text{if } ((t^Q{}_{[})^{PP} + 1)^P \subseteq t^Q \\ undefined & \text{otherwise} \end{cases}$$

**Case** *$inclusion = bigger\_part\_inside$:*

$$n' \begin{cases} (t^Q{}_{[})^{PP} & \text{if } |(t^Q{}_{[})^P \cap t^Q| \geq |(t^Q{}_{[})^P \setminus t^Q| \\ (t^Q{}_{[})^{PP} + 1 & |((t^Q{}_{[})^{PP} + 1)^P \cap t^Q| \geq |((t^Q{}_{[})^{PP} + 1)^P \setminus t^Q| \\ undefined & \text{otherwise} \end{cases}$$

**Case** *$inclusion = overlap$:*

$$n' \ (t^Q{}_{[})^{PP}.$$

*The auxiliary function 'gaps' counts the number of partitions between $n$ and $k$ which are labelled 'gap'*

$$gaps(n,k) \ \Sigma_{i=n}^{k} \begin{cases} 1 & \text{if } L_P(i^P) = \text{'gap'} \\ 0 & \text{otherwise} \end{cases}$$ ■

### 4.5.3 Extract

We define two 'extract' functions. Both extract certain parts of given intervals. The first one, 'extractL' extract partitions from given intervals which are labelled with a given label. For example, if the days in the day-partitioning are labelled 'Monday', 'Tuesday' etc., one can use extractL to extract all Tuesdays from a given interval. The parameters are $extractL(I, label, P, inclusion, intersection)$. $I$ is the interval, $label$ is the label and $P$ is the labelled partitioning. '*inclusion*' is one of the keywords 'subset', 'bigger_part_inside' or 'overlaps'. It controls the relationship between the interval $I$ and the partition $p$ with the right label. If, for example, $inclusion = subset$ then $p$ must be a subset of the support of $I$. '*intersection*' is a Boolean flag. It controls whether the result consists of just the partitions, or the partitions intersected with $I$.

**Definition 4.5.4 (extractL)** *Let $I$ be an interval, 'label' a label, $P$ a labelled partitioning with labelling $L_P$, 'inclusion' one of the key words 'subset', 'bigger_part_inside' or 'overlaps' and 'intersection' a Boolean flag.*

*We define the function extractL:*

$extractL(I, label, P, inclusion, intersection)$

$$\bigcup_{p \in P, L(p)=label, C(p)} \begin{cases} p \cap I & \textit{if } intersection = true \\ p & \textit{otherwise} \end{cases}$$

*where*

$$C(p) \begin{cases} p \subseteq S(I) & \textit{if } inclusion = \textit{'subset'} \\ |p \cap S(I)| \geq |p \setminus S(I)| & \textit{if } inclusion = \textit{'bigger\_part\_inside'} \\ p \cap I \neq \emptyset & \textit{otherwise} \end{cases}$$ ■

The next function, '*extractD*' can extract a subinterval of a certain *duration* from an interval. It can, for example, be used to extract the second fortnight in a year. The parameters are

$extractD(I, O, D, componentwise, boundaries, forward, inclusion, intersection)$

$I$ is the interval from which subintervals are to be extracted. $O$ (for offset) is a duration. It determines the offset for the interval to be extracted. For example, if the second fortnight is to be extracted, we need an offset of one fortnight. $O = ((2, week))$ could be used in this case. $D$ is the duration of the interval to be extracted. Both, the offset, and the duration must be positive. Otherwise the extractD function is undefined. *componentwise* is a flag. If it is false then only one single subinterval is extracted. If it is true then a subinterval of each component of $I$ is extracted. '*boundaries*' is one of the keywords 'support', 'core' or 'kernel'. It determines the part of the interval $I$ which is to be used for the extraction. $forward$ is a Boolean flag. If it is true then the offset is computed from the start of the interval $I$. Otherwise it is computed from the end of the interval. '*inclusion*' is one of the keywords 'align', 'subset', 'bigger_part_inside' or 'overlaps'. It is used in the functions *startpoint* and *endpoint* and determines exactly from where the offset is computed. In the above example, one would choose $inclusion = bigger\_part\_inside$. This causes that the usual convention that the first week in a year (or any other interval) is the week whose bigger part is in the interval. '*intersection*' is again a Boolean flag. If it is true then the time interval which is determined by the offset and the duration is intersected with $I$. Otherwise this time interval is returned, even if its intersection with $I$ is empty. This feature can, for example, be used to compute the second fortnight from now. It is sufficient to have an interval $I$ which is just one second long. With the flags $inclusion = align$ and $intersection = false$ one would get the 2 weeks interval.

The *extractD* function needs two auxiliary functions *startpoint* and *endpoint*. They compute the exact start and endpoint for the offset.

**Definition 4.5.5 (startpoint, endpoint)** *let $t$ be a time point, $P$ a partitioning and 'inclusion' one of the keywords 'align', 'subset', 'bigger_part_inside' or 'overlap'. We define the two functions*
*$startpoint(t, P, inclusion)$ and $endpoint(t, P, inclusion)$:*

$$\begin{array}{ll}
startpoint(t, P, \text{`}align\text{'}) & t \\
startpoint(t, P, \text{`}subset\text{'}) & \begin{cases} t^{P}{}_{[} & \textit{if } t = t^{P}{}_{[} \\ t^{P}{}_{]} & \textit{otherwise} \end{cases} \\
startpoint(t, P, \text{`}bigger\_part\_inside\text{'}) & \begin{cases} t^{P}{}_{[} & \textit{if } t - t^{P}{}_{[} \leq t^{P}{}_{]} - t \\ t^{P}{}_{]} & \textit{otherwise} \end{cases} \\
startpoint(t, P, \text{`}overlap\text{'}) & t^{P}{}_{[} \\
 & \\
endpoint(t, P, \text{`}align\text{'}) & t \\
endpoint(t, P, \text{`}subset\text{'}) & \begin{cases} t^{P}{}_{[} & \textit{if } t = t^{P}{}_{[} \\ (t^{PP} - 1)^{P}{}_{[} & \textit{otherwise} \end{cases} \\
endpoint(t, P, \text{`}bigger\_part\_inside\text{'}) & \begin{cases} t^{P}{}_{]} & \textit{if } t - t^{P}{}_{[} \geq t^{P}{}_{]} - t \\ t^{P}{}_{[} & \textit{otherwise} \end{cases} \\
endpoint(t, P, \text{`}overlap\text{'}) & \begin{cases} t^{P}{}_{[} & \textit{if } t = t^{P}{}_{[} \\ (t^{P}{}_{]} & \textit{otherwise} \end{cases}
\end{array}$$

■

The next auxiliary function $intersect(i, I, inclusion, intersection)$ checks whether the interval $i$ is an admissible part of the result of *extractD*. This depends on the control parameter '*inclusion*' which determines the relationship between $i$ and the original interval $I$. The second parameter '*intersection*' controls whether $i$ itself should be the result, or the intersection of $i$ with $S(I)$.

**Definition 4.5.6 (intersect)** *Let $i = [t_1, t_2[$ be a crisp interval, and let $I$ be a fuzzy interval. Let 'inclusion' be one of the keywords 'subset', 'bigger_part_inside' or 'overlap'. Let 'intersect' be a Boolean flag.*
*We define*
$intersect(i, I, inclusion, intersection)$

$$\begin{cases} i \cap S(I) & \textit{if } keep = true \textit{ and } intersect = true \\ i & \textit{if } keep = true \textit{ and } intersect = false \\ \emptyset & \textit{otherwise} \end{cases}$$

*where*

$$\begin{array}{lll}
keep = true & \textit{iff} & inclusion = \text{`}subset\text{'} \textit{ and } i \subseteq S(I) \textit{ or} \\
 & & inclusion = \text{`}bigger\_part\_inside\text{'} \textit{ and } |i \cap S(I)| \geq |i \setminus S(I)| \textit{ or} \\
 & & inclusion = \text{`}overlap\text{'} \textit{ and } i \cap S(I) \neq \emptyset
\end{array}$$

**Definition 4.5.7 (extractD)** *Let $I$ be an interval, let $O = ((n_0, P_0), \ldots)$ (for offset) and $D$ (for duration) be two durations. 'componentwise', 'forward' and 'intersection' are three Boolean flags. 'boundaries' is one of the keywords 'support', 'core' or 'kernel' and 'inclusion' is one of the keywords 'align', 'subset', 'bigger_part_inside' or 'overlaps'.*
*We define the function*
$extractD(I, O, D, componentwise, boundaries, forward, inclusion, intersection)$
*as follows:*

*Let* $[a, b[ \begin{cases} [I^{fS}, I^{lS}[ & \textit{if boundaries} = \textit{'support'} \\ [I^{fC}, I^{lC}[ & \textit{if boundaries} = \textit{'core'} \\ [I^{fK}, I^{lK}[ & \textit{otherwise} \end{cases}$

*be the relevant boundaries for extracting the interval. The extractD function is undefined if one of the boundaries is infinite.*

*Let* $t_0 \begin{cases} \textit{startpoint}(a, P_0, \textit{inclusion}) & \textit{if } forward = true \\ \textit{endpoint}(b, P_0, \textit{inclusion}) & \textit{otherwise} \end{cases}$

$t_0$ *is the start point for computing the offset.*

*Let* $[t_1, t_2[ \begin{cases} [shift(t, O), shift(t_1, D)[ & \textit{if } forward = true \\ [shift(t_2, -D), shift(t, -O)[ & \textit{otherwise} \end{cases}$

*be the interval to be extracted.*

$\textit{extractD}(I, O, D, false, \textit{boundaries}, forward, \textit{inclusion}, intersection)$
$\quad intersect([t_1, t_2[, I, \textit{inclusion}, intersection).$

*extracts one subinterval from* $I$.

$\textit{extractD}(I, O, D, true, \textit{parameters})$
$\begin{cases} \bigcup_{J \in Cmp(I)} \textit{extractD}(J, O, D, false, parameters) \\ \quad \textit{if componentwise} = \textit{true} \\ \textit{extractD}(I, O, D, false, parameters) \quad \textit{otherwise} \end{cases}$

*extracts a subinterval from each component.*
*'parameters' stands for 'boundaries,* $forward$*, inclusion,* $intersection$*'.* ■

A simplified version of '*extractD*' needs no duration $D$, but only an offset $O$, and extracts instead of an interval, just the start point of the interval.

## 4.6 Split

The function 'split' splits an interval into several intervals of a given duration. As an example for its use, suppose we want to define a function 'Weekend(I)', which extracts from the interval $I$ all weekends. One way to do this would be to partition the interval $I$ into intervals of one week length, and then to use the function *extractD* to extract from each week the sixth and the seventh day. The function

$\textit{split}(I, D, \textit{componentwise}, \textit{boundaries}, forward, \textit{inclusion}, \textit{sequencing},$
$\quad intersect)$

is controlled by a whole bunch of parameters. The flag '*componentwise*' indicates whether the interval $I$ is split as a whole, or whether its components are split separately. The keyword '*boundaries*' determines whether the support of $I$, the core of $I$ or the kernel of $I$ is to be split. The flag '$forward$' indicates whether the interval is to be split from left to right or from right to left. *inclusion* is a further keyword. It controls where to start with the splitting and determines the relationship between the split parts and the interval $I$. For example, if $\textit{inclusion} = \textit{subset}$ then all split parts must be a subset of the support if $I$. The keyword '*sequencing*' becomes relevant when the duration $D$ is not a single partition, but something mixed. If, for example, $D = (1week, 3day)$ we want to split $I$ into intervals of length 1 week plus 3 days. There are three different options. The first possibility is to partition $I$ sequentially without gaps. The second possibility is to synchronize the split parts with the 'week' partitioning. That means each split part starts with the start of a week. Since the length of the split parts is not a multiple of a week length, we must either leave gaps (second possibility) or compute overlapping split parts (third possibility).

Finally the '*intersect*' flag controls whether the sequence of partitions is the result of split, or whether the partitions are intersected with $I$.

The 'split' function needs two auxiliary functions, '*advance*' and '*retract*'. Depending on the value of the keyword 'sequencing' they compute the start point of the next split part. '*advance*' is used for forward splitting and '*retract*' is used for backward splitting.

**Definition 4.6.1 (advance and retract)** *Let $t$ be a time point, $P$ a partitioning and sequencing one of the key words 'sequential', 'overlap' or 'with_gaps'. We define the function 'advance':*

$$advance(t, P, sequencing) \begin{cases} t & \text{if sequencing} = \text{'sequential'} \\ t^P{}_{[} & \text{if sequencing} = \text{'overlap' or } t = t^P{}_{[} \\ t^P{}_{]} & \text{otherwise} \end{cases}$$

*We define the function 'retract:*

$$retract(t, P, sequencing) \begin{cases} t & \text{if sequencing} = \text{'sequential'} \\ t^P{}_{]} & \text{if sequencing} = \text{'overlap' or } t = t^P{}_{]} \\ t^P{}_{[} & \text{otherwise.} \end{cases}$$

■

**Definition 4.6.2 (split)** *Let $I$ be an interval and $D = ((n_0, P_0), \ldots)$ be a duration. Let 'boundaries' be one of the keywords 'support', 'core' or 'kernel'. Let 'componentwise', 'forward' and 'intersect' be Boolean flags. Let 'inclusion' be one of the keywords 'subset', 'bigger_part_inside' or 'overlap', and let 'sequencing' be one of the key words 'sequential', 'overlap' or 'with_gaps'.*

*We define the function split:*

*$split(I, D, componentwise, boundaries, forward, inclusion, sequencing,$ $intersect)$ which computes a list $(S_0, \ldots)$ of intervals.*

**Case componentwise = false***:*

$$\text{Let } [a, b[ \begin{cases} [I^{fS}, I^{lS}[ & \text{if boundaries} = \text{'support'} \\ [I^{fC}, I^{lC}[ & \text{if boundaries} = \text{'core'} \\ [I^{fK}, I^{lK}[ & \text{otherwise} \end{cases}$$

*be the relevant boundaries for splitting the interval. The split function is undefined if one of the boundaries is infinite.*

$$\text{Let } t_0 \begin{cases} startpoint(a, P_0, inclusion) & \text{if } forward = true \\ endpoint(b, P_0, inclusion) & \text{otherwise} \end{cases}$$

*be the start point for computing the split.*

*We compute a first sequence $S'_i$ of intervals:*

*if $forward = true$ then*

*$S'_0$ $[t_0, shift(t_0, D)[$ and $S'_{i+1}$ $[s, t[$ where $s$ $advance(S'_{i]}, P_0, sequencing)$ and $t$ $shift(s, D)$.*

*The sequence $S'_i$ is computed until $S'_{i]} \geq b$.*

*if $forward = false$ then*

*$S'_0$ $[shift(t_0, -D), t_0[$ and $S'_{i+1}$ $[s, t[$ where $t$ $retract(S'_{i[}, P_0, sequencing)$ and $s$ $shift(t, -D)$.*

*The sequence $S'_i$ is computed until $S'_{i[} \leq a$.*

*The final sequence* $(S_0, \ldots)$ *is now obtained from the* $S'_i$ *by computing*

$S_i \;\; intersect(S'_i, I, inclusion, intersect)$

*and removing the empty intervals from the sequence.*

**Case componentwise = true***:*
$split(I, D, true, parameters) \;\; \bigcup_{J \in Comp(I)} split(J, D, false, parameters)$
*where* $parameters$ *stands for*
*'*$boundaries, forward, inclusion, sequencing, intersect$*'.*

$\bigcup$ *does in this case not mean set union, but appending the lists of intervals, and removing duplicates.* ■

An appication for the '*split*' function is the representation of clock times. For example, the interpretation of '8:30 am' could be the set of all time points corresponding to 8:30 am. Alternatively it could mean a function mapping an interval $I$ to the time points in $I$ which corresponds to 8:30 am. This could be achieved as a composition of the '*split*' function and the '*extract*' function.

## 4.7 Summary

In this chapter a number of basic operations have been defined which operate on time points, crisp and fuzzy time intervals, partitionings and durations. Together with the functions and relations defined for each of these datatypes separately, they form the building blocks for a powerful specification language for temporal notions. This language can then be used as a computational semantics for temporal expressions in structured and semistructured data, and for XML and database query languages.

The functions at the mixed function layer were chosen to be *minimal* and *powerful*. Minimal means that they cannot be decomposed into simpler functions. Powerful means that as many different operations on fuzzy time intervals as possible can be defined by combining the mixed functions and the functions at the lower layer. So far, this is not such a formal notion as the general notion of computable functions. Experience and practical applications will show whether other functions need to be added.

The functions will be available in the WebCal system. This system is developed for processing complex temporal notions as realistically as possible, i.e. without any approximations and abstractions. A prototype, which does not yet contain all the functions proposed in this chapter is currently being tested.

# Chapter 5

# Systematics and Architecture for a Resource Representing Knowledge about Named Entities

**Klaus U. Schulz and Felix Weigel**

**Abstract**

Named entities are ubiquitous in documents in the web and other document repositories. The information that a human user associates with named entities occurring in a document often suffices to derive a simplified picture, or a fingerprint, of its contents. Quite generally, background knowledge on named entities simplifies proper document understanding. In order to use this kind of information in automated document processing, resources are needed that make information implicitly carried by named entities explicit, formalizing it in an appropriate way. We describe the systematics and architecture of an experimental resource that contains a thematic-geographic-temporal hierarchy for classifying named entities, positions named entities of various kinds with respect to the hierarchy, lists synonyms, and gives formal descriptions of these entities and their relations. The resource should offer a general basis for semantic annotation, indexing, retrieval, querying, browsing and hyperlinking of (semi-)textual web documents, structured documents and flat texts.

## 5.1 Introduction

Named entities of various categories, such as “Ludwig van Beethoven”, “Daimler-Chrysler”, “Dresdner Bank”, “Kofi Annan”, “Second World War”, “Coca-Cola”, “International Conference on Logic Programming”, “February”, “Royal Albert Hall”, “Niagara Falls” etc. are ubiquitous in (semi-)textual documents, including documents in the web. Typically they carry a large amount of implicit semantic information and associations that help human users to get a simplified picture of the contents of the document. From the occurrence of “Beethoven”, “Royal Albert Hall”, “February 7, 2004”, “Fidelio” we can guess that the document announces or describes a classical concert in London. Documents with occurrences of “Beckenbauer”, "FIFA" and "World-Championship 2006" are likely to describe some current political affair in the world of football. Documents with occurrences of "Putin" and "George W. Bush" in general are related to the field of global politics. Hence, named entities occurring in a document yield an interesting fingerprint of its general content. Furthermore, in order to "understand" contents or meta-information of a given document it might be necessary to know, say, that "Regensburg" is a town in "Bavaria/Germany" and "Christmas" is in "December".

Currently, techniques for automated analysis of documents only start to make use of this kind of information. In the area of information retrieval (IR), sometimes special dictionaries for important categories of named entities (e.g., persons, geographic entities) are used. However, the adequate role of these entities in the general process of indexing and similarity search is not yet clarified in a satisfactory way. Going beyond classical IR, analysis of named entities could obviously support various ambitious document understanding/processing tasks such as, e.g., extraction of meta-information, deduction, "semantic" document transformation, automated hyperlinking and others. However, not much of this potential role is realized in actual systems.

As a first step towards an automated analysis of named entities, a more precise picture of the semantic information that comes with specific kinds of named entities has to be developed. In this chapter we focus on four general types of information. First, any named entity is associated with a collection of *thematic fields.* Second, many named entities (e.g., "Olympic Games", "International Conference on Logic Programming") are associated with a *temporal period*, or a collection of temporal periods. Third, named entities are often related to a *geographic location*, or to a collection of places. Fourth, characteristic *relations* exist between named entities of various categories (e.g., "Leonardo da Vinci" is the creator of the painting "Mona Lisa" which is located in the "Louvre"). As a matter of fact, these relations may be temporarily restricted.

Before we may use semantic information associated with named entities for document understanding/processing and reasoning tasks we have to make it explicit. A special resource is needed where information on named entities of the above type is formalized and encoded in an appropriate way. Both design and construction of such a knowledge base are difficult tasks. From a conceptual point of view a suitable hierarchical structure has to be found where thematic areas, temporal periods, locations and entities are ordered using a well-defined set of meaningful semantic relations. Using this scheme, knowledge on real world entities and topics has to be formalized and imported. From a practical point of view, the number of relevant entities is huge and associated with all kinds of thematic areas. In a way, the intended resource covers an important part of encyclopedic knowledge. On this background it makes sense to distinguish three tasks, design of a suitable architecture, realization (filling) of the resource, and adaption to specific applications.

In this chapter we concentrate on the first task. We describe our actual picture of an architecture for a resource that encodes the above-mentioned types of information for named entities. An experimental version of the resource based on this architecture is currently realized in our group, concentrating on entities in non-scientific "common-sense" thematic areas. The resource is structured using three levels. Level 1, the *navigational level*, positions named entities of various categories in a thematic-geospatial-geotemporal hierarchy. Each named entity, as well as every field in the hierarchy, comes with a short formal description including main name, synonyms and others. The ordering structure of the hierarchy is induced by meaningful operations for refining and combining entries. Level 2, the *logical level*, describes relations between entities in the hierarchy. It refines and complements the relationships implicitly encoded in the hierarchy, adding explicit relations in the form of datalog facts. Level 3, the *linguistic level*, is used to associate natural language words and phrases with the entries of the hierarchy. This should support indexing mechanisms that include thematic information as well as geospatial and temporal information. Our ideas for Level 3 are only preliminary and we largely ignore this level here.

The chapter is structured as follows. Taking the experimental nature of the resource into account, the following section collects some pictures and ideas that guide the design of the resource. In Section 5.3 we describe syntax and meaning of the identifiers that are used for the thematic-geographic-temporal hierarchy and show how the ordering relations of the hierarchy are derived from the set of identifiers of entries. Section 5.4 briefly describes the structure of a single entry of the hierarchy. Section 5.5 indicates how relations between named entities are described in Level 2 using facts. Section 5.6 adds some concluding remarks.

## 5.2 Motivation: Goals, pictures, and intended applications

In the following description, which explains the motivation behind the development of the resource in more detail, we distinguish between

- general scientific goals that are associated with the design and construction of the resource,
- pictures and ideas that influenced the design, and
- intended applications of the resource.

It should be stressed that the general development is not governed by a fixed set of intended applications. However, some ideas of future applications are needed to guide design principles.

*Goal 1. Systematics for thematic areas, periods and locations with a semantic meaning.* By *systematics* we mean an abstract ordering scheme for positioning entries of a structured knowledge base, together with a scheme for assigning *identifiers* to the entries that reflect the organization. In the area of documentation languages many distinct systematics can be found for organizing collections of thematic fields in a structured way. Most systematics yield a tree structured classification scheme that is based on a single topic/subtopic ordering relation (e.g. ACM Computing Classification Scheme [4]). Other systematics, such as the universal decimal classification [51], are more complex, but not very explicit as to the meaning of syntactic constructs used for composing identifiers. We would like to have a "typed" systematics that reflects the difference between main categories of entries (thematic fields, temporal periods, locations, entities, classes of entities, etc.). It should be possible to combine entries of distinct categories in a flexible way (e.g., ⟨"Germany","Politics"⟩ $\mapsto$ "German Politics") and to refine thematic fields and other entries (e.g., "Politics" $\mapsto$ "Foreign Politics", "Germany" $\mapsto$ "Bavaria"). Refinement and combination operations should have a defined *semantic meaning.* Further desirable properties are the following. *Stability:* local changes (e.g., addition of subentries, deletions) should not affect other parts of the identification scheme. *Arbitrary depth and branching degree:* the identification scheme should allow an arbitrary number of levels for organizing concepts and topics. Similarly it should be possible to have an arbitrary number of immediate subtopics for a given topic. *Multiple parents:* the scheme should not be restricted to tree structures. It should be possible to consider an entry as a child of distinct parent topics. *Multiple subdivision of entries* should be supported. Many entries can be further subdivided using distinct criteria (e.g., the category of politicians can be subdivided by gender, nationality, administrative role, etc.) It should be possible to encode the criterion that gives rise to a certain division into subentries. *Efficient computational access* to subordinate and superordinate concepts should be supported.

*Goal 2. Providing a basis for geospatial and geotemporal reasoning.* The data contained in the resource should help to realize simple forms of geospatial and geotemporal reasoning. To this end, the *locational or temporal position* of entities and objects should be described. For example, the resource should make clear that "Munich" is part of "Bavaria", which is part of "Germany". It should encode that "Christmas" is in "December" and "December" is in the "Second Half of a Year". A reasoning component may then conclude that a given event in Munich in December 2004 is an event in Germany in the second half of 2004. In a similar way the resource should provide basic numeric information on distances that can be used to deduce, say, that the town "Holzkirchen" is within a 50 km neighbourhood of "Munich". To this end, temporal periods ("Second World War") and locations ("Munich"), as well as other classes of entities (e.g., conferences, concerts and other events) are described with the help of calendar dates and geo-coordinates.[1] In addition we use symbolic temporal and spatial relations such as inclusion or overlap. From a practical point of view, a large set of entities carrying temporal and/or locational information has to be covered.

*Goal 3. Connecting textual expressions and vocabulary with thematic fields.*

As one vision going beyond analysis of named entities would like to be able to associate with any content word of a given dictionary a set of topical areas or domains of the hierarchy where the word has an important status. For example, the noun "heart" typically points to a small selection of topics such as "Love", "Popsongs", and "Medicine". We plan to realize a partial mapping from words/phrases to typical topics. This is the role of the above-mentioned Level 3.

*Picture 1. A logical model of prominent entities of the world.* From one point of view, with Levels 1 and 2 we would like to describe a simplified formal model of a part of the "real world" that includes time periods, places, individuals of distinct type, their relations etc. The model does not try to capture "deep knowledge" such as aspects of causality. It "merely" represents a (partial) collection of relational facts. Still, many details of this picture remain to be clarified. For example it is not really clear how the intuitive notion of a "thematic area" should be modelled in logical terms.

*Picture 2. Chaining of associations.* Any real-world topic or entity can be seen as a "mental container" for a whole class of related topics, places, periods and entities. Opening this container, or looking at it with a magnifying glass, we find new subordered topics etc. With "chaining of associations" we mean a mental process where we start with a topic or entity, move to a subtopic, find new subtopics and continue up to a certain point. One characteristic feature of chaining is that it is possible to reach a given topic on distinct paths. As an example, consider the event "Olympic Games Munich 1972". One chain, starting from "Munich" could have the form

> Munich → History of Munich → Munich in the 1970ies → Events in Munich in the 1970ies → Sport Events in Munich in the 1970ies → Olympic Games Munich 1972

Further chains leading to "Olympic Games Munich 1972"could start, e.g., from "Sports", or from the period "1970ies". When browsing the hierarchy we would like to support chaining

[1] As a starting point we intend to describe locations numerically using a single pair of geo-coordinates, marking a central point within the location. Clearly it is desirable to integrate more precise descriptions.

of associations. It should be possible to find a specific entity or topic starting from distinct points. As an immediate consequence, our hierarchical structure cannot be tree-formed. Still we definitely want to avoid cycles. When finding an entry of the hierarchy such as “Olympic Games Munich 1972” in a document we would like to be able to derive superordinate entries (“Munich”, “Sports”, “1970ies”). Once we have cycles, the notion of “superordinate” entries does not make sense.

*Intended application 1. Semantic annotation and semantic indexing of web-documents and document repositories.* With “annotation” we mean a process where pieces of meta-information are attached to a document, or to specific elements of a document. Based on the planned resource we may, e.g., enumerate named entities of a particular type found in the document/element in a syntactically normalized way, attach thematic areas, temporal periods and locations associated with these entities to documents/elements. Furthermore, Level 3 of the resource helps to map arbitrary words and phrases occurring in the document to thematic areas, and to use this mapping for semantic annotation. With “indexing” we mean the related process where we create a simplified description for each document in a given repository. We may use the resource to create specialized subindexes that (conceptually) map named entities, thematic areas, time periods and locations of the hierarchy to documents in the repository. Semantic annotation and semantic indexing support the following practical applications.

*Intended application 2. “Semantic” document querying and transformation.* A popular vision is that future generations of query/transformation languages for data on the web are equipped with special mechanisms for dealing with semantic information of distinct types. Using semantic annotations of the above-mentioned form, such languages could be able to access, query and transform documents based on topical, temporal and locational conditions. A possible web-inquiry could be: “Find web pages that mention transport enterprises in a neighbourhood of 100 km around Munich. Order them by distance from Munich.”

*Intended application 3. Semantic Retrieval.* Semantic retrieval is similar, but does not involve transformation steps. Locational, temporal or topical conditions in queries bring IR closer to data base querying. Possible retrieval tasks to be supported are, e.g. “Find documents that mention international jazz musicians and locations in Bavaria.”, or “Find documents that mention banks in Bavaria, contain the keyword ’mortgage’, and refer to each of the last twelve months.”

*Intended application 4. Typed hyperlinking.* Based on the resource we may introduce links between web-pages that support special navigation/mining tasks. We may, say, link documents that refer to the same entity of category X (person, event, film, composition, organization,...), or to some entity in a given thematic area (jazz in the 1970ies). We may use the relations of Level 2 to link documents that mention at least two compositions of Ludwig van Beethoven. Using the hierarchy, links can be typed, say, with topics like “Music”, “Politics” etc.

*Intended application 5. Browsing.* The hierarchy will include a small collection of web-addresses that are relevant for a given entry. Hence it can be used as a special (limited) kind of web-directory. Based on the above-mentioned indexing mechanisms we may also use the hierarchy to support document retrieval by interactive browsing techniques. For example, during a retrieval session where we use conventional keyword querying we may open a parallel window where we browse through the hierarchy. We may restrict a given large result set to those documents that mention at least one entity of the special category X of the currently visited entry of the hierarchy. From more general point, the knowledge available in the hierarchy is used to reduce complexity of searching and answer navigation.

## 5.3 Systematics and Organization

Thematic fields and domains, as well as geographic regions and temporal periods, can be organized in at least two conceptually distinct ways. Following an *analytical organization scheme*, topics and entities are ordered using a logical perspective. Consequently, concepts that have the same analytical status are introduced at the same level, or depth. On the other hand we may use a *relevance-based* perspective. This would mean that we try to have "important" topics close to the root of the hierarchy. As a matter of fact, the notion of "importance" is relative and depends on a given application. For example, assume that the hierarchy is used to classify news in Germany. When following an analytical ordering, the two states "Germany" and "Portugal" should be on the same level: both are European states. When using a relevance-based ordering, "Germany" has to be found on a higher level than Portugal. With our systematics we would like to support a twofold organization with an analytical principal ordering and a relevance-based secondary ordering.

### 5.3.1 Syntax of identifiers

We use the following general philosophy. Each entry of the hierarchy has a unique identifier. Identifiers have a well-defined term structure. The syntactic structure of the identifier of an entry reflects its analytical position in the hierarchy: all analytical parent- and ancestor-relations are derived from the term structure of identifiers, in a way to be explained.

In order to support a secondary relevance-based organization we may specify for a given entry a set of "godfather entries". In the visual representation we do not only enumerate the analytical children of an entry but also its godchildren. In this way an entity like "David Beckham", who might perhaps be introduced as an "Current British Football Player of Real Madrid" at a deep level of the analytical hierarchy, can be made directly visible, say, under "Sportsmen". The use of godfather parents is not only relevant for browsing. For classification tasks it might be important to know that David Beckham belongs to a small group of prominent sportsmen, which implies that documents mentioning David Beckham are related to sports, but not necessarily to the actual equipe of Real Madrid.

**Definition 5.3.1** *There are seven* basic types of identifiers*, $E$ (category of entities), $e$ (individual entity), $G$ (category of geographic areas), $g$ (individual geographic areas), $T$ (category of temporal periods), $t$ (individual temporal period), $F$ (thematic field). With $\mathcal{T} = \{E, e, G, g, T, t, F\}$ we denote the set of basic types.*

**Example 5.3.2** *Entity categories (type $E$) can be considered as unary predicates that refer to a set of entities. Examples are "Persons", "Composers", "Organizations", "Paintings". Examples for individual entities (type $e$), which can be interpreted as individual constants, are "George W. Bush", "Beethoven", "Picasso", "BMW", "The Rolling Stones". Possible categories of geographic areas (type $G$) are, e.g., "Continents", "States", "Industrial Regions". Individual geographic areas (type $g$) are, e.g., "Austria", "Germany", "Pacific Ocean", "Alps". Categories of temporal periods (type $T$) are, e.g., "Political Epochs", "Epochs in Art", "Centuries". Individual temporal periods (type $t$) are, e.g., "Middle Ages", "Second World War", "1970ies". Thematic fields (type $F$) are "Politics", "Arts", "Music", "Mathematics and Music in Ancient Greece".*

**Definition 5.3.3** *The set of* possible identifiers*, $\mathcal{I}(\mathcal{T}, \mathbb{N})$, is recursively defined as follows:*

*1.* Root Identifier. *The empty sequence, written ( ), is a possible identifier.*

*2.* Local Introduction. *If $\varphi$ is an identifier, $n$ is a positive integer, and if $X \in \mathcal{T}_0$ is a basic type, then $(X\varphi.n)$ is a possible identifier.*

*3.* Symmetric Intersection. *If $\varphi$ and $\psi$ are possible identifiers, then $(\varphi\&\psi)$ is a possible identifier. The operator "&" is considered to be associative, commutative and idempotent. Hence expressions $(\varphi_1\&\varphi_2\&\dots\&\varphi_n)$ are well-formed. The root identifier ( ) is treated as a neutral element w.r.t. "&", which means that expressions $(\varphi\&(\ ))$ and $\varphi$ are equivalent.*

*4.* Focus. *If $\varphi$ and $\psi$ are identifiers, and if $\varphi$ has type $E$, $G$, $T$, or $F$, then $(\varphi{:}\psi)$ is an identifier. The root identifier ( ) is treated as a right neutral element w.r.t. ":", which means that expressions $(\varphi : (\ ))$ and $\varphi$ are equivalent.*

We use the following *notational conventions*: If $\varphi = (\ )$ we write $(X.n)$ for $(X\varphi.n)$. Expressions $(X\varphi.n.k)$ are shorthand for $(X(X\varphi.n).k)$. Similarly $(X.n.k)$ is shorthand for $(X(X.n).k)$.

**Remark 5.3.4** *In order to account for the above equivalences, identifiers are* normalized *in the actual system. This means that nested symmetric intersections are flattened in the obvious way. Conjuncts "( )" are suppressed, as well as multiple conjuncts, the remaining conjuncts are ordered lexicographically. For example, the normal form of $(((\ )\&(F.1))\&(((F.3)\&(F.2))\&(F.3))$ is $((F.1)\&(F.2)\&(F.3))$. Furthermore, any focus on ( ) is erased.*

In the sequel, we write $\mathcal{I}^n(\mathcal{T}, \mathbb{N})$ for the set of normalized identifiers. If not mentioned otherwise, with an identifier we always mean a normalized identifier.

**Remark 5.3.5** *For points deeply embedded in the hierarchy, identifiers tend to be notationally complex and long. In the implemented version, each entry is mapped to a unique integer that represents a second, numerical identifier. An identifier of the form $(\varphi\&\psi)$ can then be written more compact in the form $(n\&m)$ where $n$ and $m$ respectively denote the numerical identifiers corresponding to $\varphi$ and $\psi$.*

### 5.3.2 Meaning of identifiers

The following remarks explain the intuition behind the above three operations and describe the meaning. Clearly, since we model parts of the real world we cannot expect to have a fully formalized semantics as, say, in mathematical logic. Still, persons that fill the hierarchy and persons that use the results should have a common picture that is as precise as possible.

**Local Introduction.** When moving to a specific topic, $\varphi$, we may find new entity classes, thematic subtopics, temporal periods, geographic areas etc. that are specific for the given topic in the sense that they do not play a major role "outside" the topic. In such a case, the entity class (or the thematic subfield, the period, the geographic area etc.) is introduced into the hierarchy using a local introduction $(X\varphi.n)$. For example, consider an entry "Music" with identifier $\varphi$. The category of entities "Compositions" is only relevant inside the area of music. We might introduce it using an identifier $(E\varphi.1)$. If $\varphi$ stands for Europe, the European Commission might be introduced as $(e\varphi.1)$, the 20 commissioners as $(E\varphi.1)$. An entry "Politics" with identifier $\psi$ might be refined using a local introduction $(F\psi.5)$ to "Foreign Politics". In contrast, assuming that "Germany" is an entry of the hierarchy, "German Politics" would not be introduced by a local introduction, but as a symmetric intersection (s.b.). The following list illustrates the formal relationship between an entry $\varphi$ and a local introduction $(X\varphi.n)$. Due to type differences there are 49 possible instances of the local introduction scheme. Hence we cannot describe all cases here. In the sequel we use superscripts to denote types. Outermost superscripts represent the type of the resulting expression.

1. $(t\varphi^t.n)^t$: a subperiod of the temporal period $\varphi$.
2. $(T\varphi^T.n)^T$: a subcategory of the category of temporal periods $\varphi$.
3. $(T\varphi^t.n)^T$: a category of temporal periods, all overlapping with temporal period $\varphi$. We do not demand containment in $\varphi$. As an example, assume that we want to introduce as a category the "Years of the Second World War" using $\varphi =$ "Second World War". Then 1945 would be a member of the category, even if the war was finished before the end of 1945.
4. $(t\varphi^T.n)^t$: a member of the category of temporal periods $\varphi$.
5. $(G\varphi^g.n)^G$: a category of locations, all overlapping with the location $\varphi$.
6. $(e\varphi^e.n)^e$: a subentity of the structured (complex) entity $\varphi$.
7. $(E\varphi^e.n)^E$: a category of entities, all subentities of the complex entity $\varphi$.
8. $(e\varphi^E.n)^e$: a member of the category of entities $\varphi$.

**Symmetric Intersection.** Given two thematic fields $\varphi^F$ and $\psi^F$ the symmetric intersection $(\varphi\&\psi)$ denotes the thematic intersection of the two fields. A symmetric intersection is used if any subtopic of $(\varphi\&\psi)$ can be considered as a common subtopic of both $\varphi$ and $\psi$, which means that $(\varphi\&\psi)$ is a subarea of both arguments. Given the analytical hierarchy defined below, $(\varphi\&\psi)$ will always be a child both of $\varphi$ and $\psi$. For unbalanced combinations where the second argument only selects a specific subarea of the first argument the focus operator is used (s.b.). In the general case the meaning of the operator "&" depends on the types of the arguments. We list some examples.

1. $(\varphi^t\&\psi^t)^t$: a temporal period representing the intersection of temporal periods $\varphi$ and $\psi$.
2. $(\varphi^t\&\psi^T)^T$: the category of temporal periods obtained by restricting category $\psi$ to those periods that overlap with period $\varphi$ (cf. Case 3 above).
3. $(\varphi^g\&\psi^G)^G$: category of locations obtained restricting category $\psi$ to those locations that overlap with location $\varphi$. For example, "Danubian states" are states overlapping with the Danube, and not states within the Danube.
4. $(\varphi^F\&\psi^t)^F$: thematic field $\varphi$ restricted to "temporal window" $\psi$. For example, if $\varphi$ denotes "Politics" and $\psi$ denotes "Second World War", then $(\varphi^F\&\psi^t)^F$ means "Politics During the Second World War".
5. $(\varphi^F\&\psi^g)^F$: thematic field $\varphi$ restricted to location $\psi$. The interpretation is liberal in the sense that subtopics only must have some strong relationship to location $\psi$. For example if $\varphi$ denotes "Politics" and $\psi$ means "Germany", then "German Politics" can be introduced as $(\varphi^F\&\psi^g)^F$. Still, events of the field "German Politics" might happen, say, in Paris, Warsaw, or Washington.

6. $(\varphi^F \& \psi^G)^F$: thematic field $\varphi$ restricted to locations of the category $\psi$. If $\varphi$ denotes "Education" and $\psi$ denotes "European States" then "Education in European States" can be introduced as $(\varphi^F \& \psi^G)^F$.

7. $(\varphi^E \& \psi^F)^E$: a subcategory of the class of entities $\varphi^E$. Only entities are considered that are relevant for the thematic field $F$ (entities "in the area" $F$).

8. $(\varphi^E \& \psi^e)^E$: a subcategory of the class of entities $\varphi^E$. Only entities are considered where entity $e$ plays a dedicated role. If $\varphi$ denotes "Symphonies" and $\psi$ denotes "Haydn" then "Symphonies of Haydn" can be introduced as $(\varphi^E \& \psi^e)^E$.

**Focus.** The focus operation $(\varphi : \psi)$ is used for a kind of combination where the first argument $\varphi$, which has type $X \in \{T, G, E, F\}$, is more privileged than the "focussed" argument $\psi$. Typically, $\psi$ represents just a kind of "object", an "orientation" of $\varphi$, the type of $(\varphi : \psi)$ is always the type of $\varphi$. For example, "French Policy concerning Germany" could be modelled as "French Policy" focussing "Germany". As a second example, consider the difference between "Political Sciences" (German word: "Politikwissenschaft") and "Policy of Science" (German word: "Wissenschaftspolitik"). The former can be modelled as "Science" focussing "Policy", the latter as "Policy" focussing "Science". Similarly "Policy of Education" would be modelled as "Policy" focussing "Education", "Political Films" as "Films" focussing "Policy". It should be mentioned that in practice we found a considerable number of cases where it is difficult to decide if a symmetric intersection or a combination using the focus operation is more appropriate. For example, under one possible interpretation, "Religious Arts" can also be considered as "Arts" focussing "Religion". The problem is to decide if icons and other paintings of the field of "Religious Arts" really have some kind of religious status. If this is typically the case, then the use of a symmetric intersection is appropriate.

**Remark 5.3.6** *For some applications it might seem desirable to model the relationship between distinct topics in a more detailed way, introducing further operations. However, we have seen above that even with our coarse scheme it is sometimes difficult to select the "correct" operation. These difficulties tend to grow with an enlarged set of operations.*

**Remark 5.3.7** *In order to design special reasoning mechanisms for the information found in the hierarchy it would be interesting to have a formal notion of a "model" of the hierarchy, purely based on algebraic notions. The development of such a notion is one point of future work. Parts of Definition 5.3.3 can be considered as a partial axiomatization. Subclasses of models could be characterized using additional axioms such as, e.g., the equivalency between* $(\varphi : (\psi_1 \& \psi_2))$ *and* $((\varphi : \psi_1) \& (\varphi : \psi_2))$.

### 5.3.3 Derived hierarchical structure

When describing a concrete thematic-geographic-temporal hierarchy we use a finite subset $H$ of the set of all normalized identifiers, $\mathcal{I}^n(\mathcal{T}, \mathbb{N})$. We now want to define an ordering relation on $H$, based on a suitable parent/child relation. It might seem natural to treat for any symmetric intersection of the form $\varphi := (\delta_1 \& \delta_2)$ the two identifiers $\delta_1$ and $\delta_2$ as analytical parents of $\varphi$. However, this leads to counterintuitive results. For example, with this choice, all identifiers of $H$ in the sequence $(\delta \& (F.1)), (\delta \& (F.1.1)), (\delta \& (F.1.1.1)), \ldots$ would be treated as unrelated children of $\delta \in H$. Thus the number of children becomes unacceptable and the intuitive internal order among these children is ignored.

**Definition 5.3.8** *The set* min-gen($\varphi$) *of* minimal generalizations *of* $\varphi \in \mathcal{I}^n(\mathcal{T}, \mathbb{N})$*, and the set of* $X$-refinements *(where* $X \in \mathcal{T}$*) of a normalized identifier is recursively defined in the following way (exponents* $n$ *denote normalization):*

1. *min-gen*$(( \ )) := \emptyset$.
2. *min-gen*$((X\psi.n)) := \{\psi\}$*; the identifier* $(X\psi.n)$ *is an* $X$*-refinement of* $\psi$.
3. *min-gen*$((\varphi\&\psi)) := \{(\varphi'\&\psi)^n \mid \varphi' \in$ *min-gen*$(\varphi)\} \cup \{(\varphi\&\psi')^n \mid \psi' \in$ *min-gen*$(\psi)\}$. *Here* $(\varphi\&\psi)$ *is an* $X$*-refinement of* $(\varphi'\&\psi)$ *iff* $\varphi$ *is an* $X$*-child of* $\varphi'$*, and similarly for* $(\varphi\&\psi)$ *and* $(\varphi\&\psi')$.
4. *min-gen*$((\varphi : \psi)) := \{(\varphi' : \psi)^n \mid \varphi' \in$ *min-gen*$(\varphi)\} \cup \{(\varphi : \psi')^n \mid \psi' \in$ *min-gen*$(\psi)\}$. *Here* $(\varphi : \psi)$ *is an* $X$*-refinement of* $(\varphi' : \psi)$ *iff* $\varphi$ *is an* $X$*-refinement of* $\varphi'$*, and similarly for* $(\varphi : \psi)$ *and* $(\varphi : \psi')$.

*With* gen($\varphi$) *we denote the set of all* generalizations *of* $\varphi$*, which is obtained using the transitive closure of the min-gen-relation.*

We define $\psi <_{gen} \varphi$ iff $\psi \in$ gen($\varphi$) and call "$<_{gen}$" the *refinement/generalization order* on normalized identifiers in $\mathcal{I}^n(\mathcal{T}, \mathbb{N})$. It is simple to see that $\psi <_{gen} \varphi$ implies that the notational length of $\psi$ is smaller than the length of $\varphi$. As a trivial consequence we obtain:

**Lemma 5.3.9** *The refinement ordering "*$<_{gen}$*" is a strict partial ordering on* $\mathcal{I}^n(\mathcal{T}, \mathbb{N})$*. It imposes the structure of a rooted directed acyclic graph on* $\mathcal{I}^n(\mathcal{T}, \mathbb{N})$.

**Definition 5.3.10** *Let* $H$ *be a finite subset of* $\mathcal{I}^n(\mathcal{T}, \mathbb{N})$*, let* $\varphi \in H$*. The set* $R \subseteq$ gen$(\varphi) \cap H$ *is called a* minimal coverage *of* $\varphi$ *w.r.t.* $H$ *iff the following conditions hold:*

1. Coverage. *For all* $\psi \in$ gen$(\varphi) \cap H$ *there exists an element* $\varphi' \in R$ *such that* $\psi \leq_{gen} \varphi'$.
2. Minimality. *If* $\varphi_1$ *and* $\varphi_2$ *are elements of* $R$*, then neither* $\varphi_1 <_{gen} \varphi_2$ *nor* $\varphi_2 <_{gen} \varphi_1$.

It is not difficult to see that a minimal coverage always exists and is unique.

**Definition 5.3.11** *Let* $H$ *be a finite subset of* $\mathcal{I}^n(\mathcal{T}, \mathbb{N})$*. Then the* analytical parents *of* $\varphi \in H$ *w.r.t.* $H$ *are the elements of the minimal coverage of* $\varphi$ *w.r.t.* $H$.

**Example 5.3.12** *We illustrate the definition of the analytical parent-child relation with an example from sports where we show how the Spanish football club "Real Madrid" could be positioned in a suitable (fragment of the) hierarchy. Let us assume we have the following entries in H:* $(F.9)$ *"Sports",* $(E.3)$ *"Organizations",* $\sigma$ *"Spain" (we leave the form of* $\sigma$ *open),* $((F.9)\&\sigma)$ *"Spanish Sports",* $((F.9)\&(E.3))$ *"Sports Organizations",* $(F.9.1)$ *"Ball Games",* $(F.9.1.1)$ *"Football",* $((F.9.1.1)\&(E.3))$ *"Football Organizations",* $((F.9.1.1)\&\sigma)$ *"Spanish Football",* $(E((F.9.1.1)\&(E.3)).1)$ *"Football Clubs",* $((E((F.9.1.1)\&(E.3)).1)\&\sigma)$ *"Spanish Football Clubs". Then* $(e((E((F.9.1.1)\&(E.3)).1)\&\sigma).1)$ *might denote "Real Madrid". The analytical parent-child relationship induced by this set-up can be seen in Figure 5.1. Now assume that we delete entry* $((F.9.1.1)\&\sigma)$ *"Spanish Football" in* $H$*. Then link (1) disappears, and links (2), (3) are merged into a link from "Spanish Sports" to "Spanish Football Clubs".*

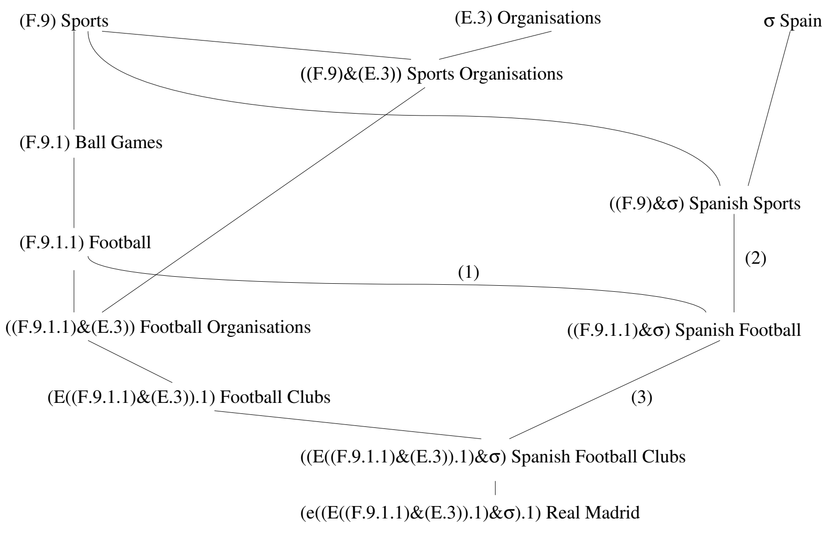

(F.9) Sports
(E.3) Organisations
σ Spain
((F.9)&(E.3)) Sports Organisations
(F.9.1) Ball Games
((F.9)&σ) Spanish Sports
(F.9.1.1) Football
(1)
(2)
((F.9.1.1)&(E.3)) Football Organisations
((F.9.1.1)&σ) Spanish Football
(E((F.9.1.1)&(E.3)).1) Football Clubs
(3)
((E((F.9.1.1)&(E.3)).1)&σ) Spanish Football Clubs
(e((E((F.9.1.1)&(E.3)).1)&σ).1) Real Madrid

Figure 5.1: Parent-child relationship derived from identifiers (cf. Example 5.3.12).

Note that the semantics of the analytical parent-child relation can be derived from the meaning of the operations. From Section 5.3.2 it follows that in Figure 5.1 $((F.9.1.1)\&\sigma)$ "Spanish Football" is a thematic subarea of $(F.9.1.1)$ "Football" obtained by looking only at those aspects/subtopics with a close relationship to the location $\sigma$ "Spain". Similarly it follows that $((F.9.1.1)\&(E.3))$ "Football Organisations" is obtained from $((F.9)\&(E.3))$ "Sports Organisations" by restricting organisations in the area of sports to organisations in the more specific area of football. Still, the ordering relations of the hierarchy only capture a part the information that is encoded in the identifiers, due to "missing points" in the hierarchy (such as "Spanish Football Organisations" above).

**Remark 5.3.13** *The actual computation of analytical parents of an identifier $\varphi$ w.r.t. a given set $H \subset \mathcal{I}^n(\mathcal{T}, \mathbb{N})$ proceeds in two steps. (1) We first compute a coverage for $\varphi$. To this end we "treat" every minimal refinement $\varphi'$ of $\varphi$. To treat $\varphi'$ means: check if $\varphi' \in H$. In the positive case, add $\varphi'$ to the coverage for $\varphi$. In the negative case, treat the minimal refinements of $\varphi'$. (2) Given a coverage $C$ we erase all members that refine other members of $C$. Details are omitted.*

The transitive closure of the analytical parent relation on $H$ is called the *ancestor-relation* on $H$ and denoted "$<^H_{gen}$".

**Definition 5.3.14** *A subset $H$ of $\mathcal{I}^n(\mathcal{T}, \mathbb{N})$ is called* constructive *iff the following conditions hold: (1) $(\ ) \in H$, (2) $(X\varphi.n) \in H$ implies that $\varphi \in H$ ($X \in \{t, T, g, G, e, E, F\}$ and $n \in \mathbb{N}$), (3) $(\varphi\&\psi) \in H$ implies $\varphi \in H$ and $\psi \in H$, (4) $(\varphi : \psi) \in H$ implies $\varphi \in H$ and $\psi \in H$.*

**Lemma 5.3.15** *Let $H$ be a constructive subset of $\mathcal{I}^n(\mathcal{T}, \mathbb{N})$. Then the ancestor relationship "$<^H_{gen}$" is a strict partial order on $H$ that imposes the structure of a rooted directed acyclic graph on $H$.*

**Remark 5.3.16** *In the graphical visualization we not only depict analytical children of a given entry, but also visualize the godchildren. As long as we do not control the introduction of godfathers there is no guarantee that we run into a loop when we follow arbitrary chains of children. For this reason, the visualization of analytical children and godchildren is distinct.*

### 5.3.4 Examples from the experimental version

In order to illustrate the systematics we add two further examples from the experimental version.

**Example 5.3.17** *The first level (analytical children of the root node) of the current hierarchy has the following entries.*[2] *(E.1) "Persönlichkeiten" (VIPs), (E.2) "Events", (E.3) "Organisationen und Einrichtungen" (Organisations and Institutions), (F.1) "Politik" (Politics), (F.2) "Wirtschaft" (Economy), (F.3) "Finanzen" (Financial Sector), (F.4) "Recht und Justiz" (Law), (F.5) "Wissenschaft und Technik" (Science and Technology), (F.6) "Kunst und Kultur" (Arts and Culture), (F.7) "Medien und Kommunikation" (Media and Communication), (F.8) "Bildung, Erziehung, Ausbildung und Beruf" (Education and Profession), (F.9) "Sport" (Sports), (F.10) "Religion", (F.11) "Lifestyle", (F.12) "Gesundheit und Ernährung" (Health and Food), (F.13) "Natur und Umwelt" (Nature and Environment), (G.1) "Geophysische Lokationen" (Geophysical Locations), (G.2) "Politische geographische Lokationen" (Political Geographic Locations), (G.3) "Kulturelle und religiöse Lokationen" (Cultural and Religious Locations), (t.1) "Geschichte" (History), (T.1) "Epochen" (Epochs), (T.2) "Jahrhunderte" (Centuries).*

**Example 5.3.18** *Figure 5.2 gives a partial picture of the hierarchy, including temporal and geographic axes, and illustrates how entities such as "Leonardo da Vinci" and "Mona Lisa" are positioned in the hierarchy.*

## 5.4 The structure of the entries

We have seen how entries are positioned in the hierarchy. In order to complete the description of Level 1 of the resource, some words on the structure of entries are in order. Any entry of the hierarchy has the following components:

1. *Identifier* (element of $\mathcal{I}^n(\mathcal{T}, \mathbb{N})$).

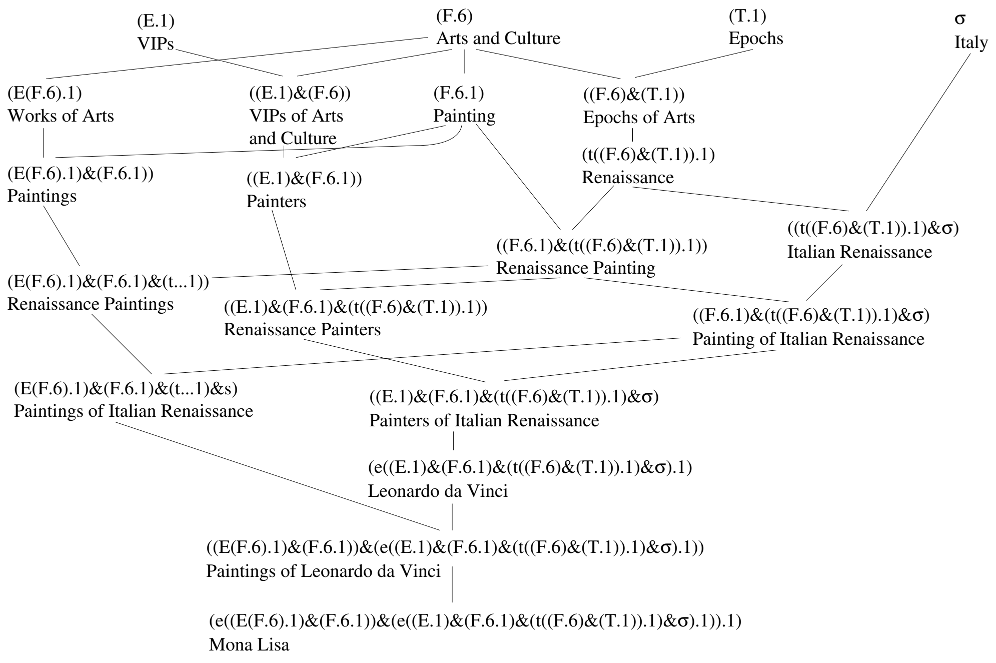
2. *Secondary identifier* (integer).

3. *Main name* (non-empty string).

4. *Explanation* (string, possibly empty). Explains the meaning/use of the main name in the present context. For human readers.

5. *Identifiers of godfather entries* (list, possibly empty). Determines entries of the hierarchy where the given entry is treated as an non-analytical immediate child since it is considered as a relevant subentry.

[2]The current version of the resource uses German notions. English translations are given in parentheses for convenience.



(E.1)
VIPs
(F.6)
Arts and Culture
(T.1)
Epochs
σ
Italy
(E(F.6).1)
Works of Arts
((E.1)&(F.6))
VIPs of Arts and Culture
(F.6.1)
Painting
((F.6)&(T.1))
Epochs of Arts
(t((F.6)&(T.1)).1)
Renaissance
(E(F.6).1)&(F.6.1))
Paintings
((E.1)&(F.6.1))
Painters
((t((F.6)&(T.1)).1)&σ)
Italian Renaissance
((F.6.1)&(t((F.6)&(T.1)).1))
Renaissance Painting
(E(F.6).1)&(F.6.1)&(t...1))
Renaissance Paintings
((E.1)&(F.6.1)&(t((F.6)&(T.1)).1))
Renaissance Painters
((F.6.1)&(t((F.6)&(T.1)).1)&σ)
Painting of Italian Renaissance
(E(F.6).1)&(F.6.1)&(t...1)&s)
Paintings of Italian Renaissance
((E.1)&(F.6.1)&(t((F.6)&(T.1)).1)&σ)
Painters of Italian Renaissance
(e((E.1)&(F.6.1)&(t((F.6)&(T.1)).1)&σ).1)
Leonardo da Vinci
((E(F.6).1)&(F.6.1))&(e((E.1)&(F.6.1)&(t((F.6)&(T.1)).1)&σ).1))
Paintings of Leonardo da Vinci
(e((E(F.6).1)&(F.6.1))&(e((E.1)&(F.6.1)&(t((F.6)&(T.1)).1)&σ).1)).1)
Mona Lisa

Figure 5.2: Part of the experimental hierarchy.

6. *Synonyms* (list of strings, possibly empty). Collects distinct natural language expressions that can be interpreted (in one reading) as synonyms of the main name. Synonyms are important for recognizing entities and topics in texts. Clearly, a remaining difficult problem is the resolution of ambiguities.

7. *Relationship to entries of other classification schemes.* Explains synonymy, hyperonymy, hyponymy or similarity relations with subject categories from other classification schemes such as the universal decimal classification UDC [51] and the IPTC subject classification [28].

In addition, all entries of type $X \in \{g, t, e\}$ come with a *formal description* that depends on the category of the entry. In the experimental version, the format of the formal description has only been fixed for a small number if entity classes. As an illustration we describe which kind of data should be included in the formal description of events. Some of these data belong to Level 2 (logical level).

1. *Main URL* (optional). If there is any URL especially for the event.

2. *Useful URLs.* URLs where useful information associated with the event can be found.

3. *Temporal description*:

   (a) Singular or periodical (sin/per).

   (b) First/last occurrence (only for periodical events). Last occurrence: use the last occurrence that is confirmed. This may be a point in future.

   (c) Turnus (for periodical events only). ($n$ days, $n$ weeks, $n$ months, or $n$ years ($n \in \mathbb{N}$).

   (d) Duration. ($n$ days, $n$ weeks, $n$ months, or $n$ years $n \in \mathbb{N}$). May be qualified with "approx.".

   (e) Temporal "home" position. For periodical events list a period that describes the usual temporal position of the event within a year as precisely as possible (upper and lower boundary). Similar for singular events. There are the following alternatives: For specifying a rough period of the year, the categories (first half, second half, spring, summer, autumn, winter) may be used. It is also possible to list an interval of months (from April to August). It is also possible to give a more precise interval such as "from April 15 to August 15".

4. *Location.* A state plus a subcountry (such as Bavaria, optional) plus a city (optional) plus an address (optional). States and cities should be introduced in the hierarchy. We use their identifiers. Exceptions are possible (small unknown towns). For periodical events with distinct places we only give a common location of all instances. (E.g., "Europe" for European championships).

5. *Location URL* (optional). This might be the URL of a concert hall, of a theatre, of a city, of a country,...

6. *Organizer* (optional). This might be another entity of the hierarchy (give identifier and main name), or a freely specified entity.

7. *Organizer URL* (optional).

8. *Importance.* International (i), national (n), or local (l).

## 5.5 Level 2: Facts

Level 2 of the planned resource represents a collection of facts that yield a more precise description of the relations between distinct entries of a hierarchy. Facts are classified into several categories.

**1. Compositional facts** describe the composition of an entry in terms of other entries.

**Definition 5.5.1** *A* symmetric union *has the form* $(\psi_1 \sqcup \ldots \sqcup \psi_n)$ *(*$n \geq 2$*) where the components* $\psi_i$ *are distinct normalized identifers of the same type* $X \in F, E, G, T, g, t$ *(*$1 \leq i \leq n$*).* $(\psi_1 \sqcup \ldots \sqcup \psi_n)$ *has type* $X$*.*

The operator "$\sqcup$" is considered to be associative, commutative and idempotent. Read, e.g., $((F.1) \sqcup (F.3))$ as "Union of universal thematic fields 1 and 3", $((G.1) \sqcup (G.3))$ as "Union of the categories of geo-entities 1 and 3", and $((g.1) \sqcup (g.3))$ as "Spatial union of the geo-entities 1 and 3", and $((t.1) \sqcup (t.3))$ as "Temporal union of the temporal periods 1 and 3".

**Definition 5.5.2** *Let* $\varphi$ *denote an identifier, let* $\alpha \in D$ *be a possible division criterion. A* compositional fact *for* $\varphi$ *has the form* $\varphi \equiv (\psi_1 \sqcup \ldots \sqcup \psi_n)$*,* $\varphi \equiv [\alpha](\psi_1 \sqcup \ldots \sqcup \psi_n)$*,* $\varphi \sqsubseteq (\psi_1 \sqcup \ldots \sqcup \psi_n)$*, or* $\varphi \sqsubseteq [\alpha](\psi_1 \sqcup \ldots \sqcup \psi_n)$ *where the* $\psi_i$ *have the same type* $X \in \{F, E, G, T, g, t\}$ *as* $\varphi$*.*

**Example 5.5.3** *Let* fmm $\in D$ *stand for the male-female distinction. Let* $(E(...).n)$ *denote any category of persons, say, politicians. Assume that we have two further entries with identifiers* $(E(...).n.k)$ *and* $(E(...).n.l)$ *denoting female and male politicians. Then the axiom* $(E(...).n) \equiv [fmm]((E(...).n.k) \sqcup (E(...).n.l))$ *expresses that the class of all politicians is the union of the classes of female politicians and male politicians, and that the division follows the male-female distinction.*

**Example 5.5.4** *Let* adm $\in D$ *stand "immediate political-administrative subunit". Then an symmetric union of the German "Bundesländer" (Baden-Wurttemberg, Bavaria, ..., Thuringia) and a compositional fact can be used to express that Germany can be partitioned into* 16 *immediate political-administrative subunits.*

**2. Pure geographic facts** express relations between entities of type $g$. *Symbolic pure geographic facts* are based on a collection of unary and binary relations between geo-entities.

1. *extended* (unary). Used for geographic areas like states.
2. *linear* (unary). Used for geo-entities like rivers and roads with a linear form on maps.
3. *point* (unary). Used for geo-entities like waterfalls, churches, ... which are represented as points on maps. Cities/towns are not treated as points.
4. *part_of* (binary). Describes spatial inclusion.
5. *common_border_line* (binary). For extended geo-entities with a common borderline such as France and Belgium.
6. *overlaps* (binary). Describes two extended geo-entities where the intersection is extended and a proper subarea of both entities. For example, Austria and the alps overlap.

7. *is-capital-of* (binary).

*Numeric pure geographic facts* assign a pair of coordinates to a geographic entity. The point described by the coordinates is meant to denote one central point of the geographic entity. Later we perhaps look at more complex spatial descriptions of extended geographic entities.

**3. Pure temporal facts** express relations between entities of type $t$. As for geographic facts, there are two kinds of pure temporal facts. *Symbolic temporal relations.* We use Allen's 13 relations for temporal intervals [5]. *Numeric temporal relations* assign a starting date and an end point to a temporal entity.

**4. Roles.** Recall from the discussion of the local introduction operation that the hierarchy in Level 1 is rich enough to encode elementship in sets represented as entries of type $X \in \{E, G, T\}$. However, the hierarchy does not systematically encode relations of arity $n \geq 2$ between entities. To this end we introduce a set $R$ of *roles* in Level 2. The following is a non-exhaustive and preliminary list. For relations with one temporal argument, this argument may either be an entry of the hierarchy or an explicit date (a century, decade, year, month, or a day).

1. *Is-location-of.* Ternary relation; first argument type $e$, second argument type $G$, third argument type $g$. Used to express locations that are stable. E.g., the location of "Hannover Messe" w.r.t. the category of locations "Towns" is "Hannover". The location of "Carnegy Hall" w.r.t. the category of locations "state" is "USA".

2. *Is-time-of.* Ternary relation; first argument type $e$, second argument type $T$, third argument type $t$. Used for entities that are naturally associated with a temporal period. E.g., the time of the "French Revolution" w.r.t. the category of "Years" is "1789".

3. *Is-the-of-in.* Arity 4. First argument type $e$, second argument type $E$, third argument type $e$ or $g$, fourth argument type $t$. With this predicate we express that a twofold restriction of an entity category determines a unique person, and that at some moment in time the person satisfied the predicate. E.g., "George W. Bush" is the "President" of the "USA" in "2003". Again we assume that "George W. Bush", "President" and "USA" are entries of the hierarchy. Similarly person $X$ might be the chief manager of an enterprise in a given year.

4. *Is-the-of-from-until.* These axioms are similar, but they specify two points in time that mark the beginning and the end of the role. Example pattern: X was president of $Y$ from $U$ until $V$.

5. *Is-a-of-in.* Similar as "Is-the-of-in"-facts, but we do not assume that the role determines a unique person. A person can be a member of a political party in a fixed year. A person can be a member of a football equipe at a moment of time.

6. *Is-a-from-until.* Similar as an "Is-the-of-from-until"-relation, but we do not assume that the role determines a unique entity.

7. *Is-the-location-of-in.* For example, "Edinburgh" is the location of the "ICDAR"-conference in "2003".

8. *Is-the-in.* "Woytila" is the "Roman Pope" in "2002".

9. *Is-the-from-until.*

## 5.6 Concluding remarks

In this chapter we introduced a systematics and a three-level architecture for a resource that encodes knowledge about named entities using a thematic-geographic-temporal hierarchy. Though the systematics described in Section 5.3 is stable, the complete picture of the resource is still preliminary in many respects. We are not aware of another resource or knowledge base with a similar functionality and structure. In our approach the thematic hierarchy - or the corresponding set of identifiers - is considered as a kind of quotient term algebra with a derivable ordering structure that represents a simplified model of real-world thematic areas, entities, and their relations. We think this perspective deserves further attention. We intend to discuss how formal models based on conventional set-theoretic notions can be used to derive a better semantics. This problem is not simple. For example, thematic areas are always described using natural language expressions. The inherent vagueness and context-adaptivity represents one major obstacle.

We only started with second big task, the explicit construction ("filling") and maintenance of the resource. Here we face a considerable number of questions and difficulties. Some of these might give rise to interesting research problems. *(1) Hierarchy construction.* Due to the interleaved nature of the hierarchy and the analytical ordering structure, which is fully derived from identifiers, persons that "fill" a subpart of the thematic hierarchy have to be aware of neighboured areas and their structure/identifiers. Hence hierarchy construction is much more difficult than for any tree-based classification scheme. *(3) Editing the hierarchy.* Intelligent techniques for editing a given hierarchy have to be developed. *(4) Collecting data.* The collection of the data that are necessary to realize the resource represents a huge amount of work. The process should be partially automated.

As a matter of fact, many particular aspects of the resource can be found in other work. A lexical treatment of domains, e.g, is discussed in [27]. Thesauri [1] or meta-thesauri [36] are obviously close to our resource. In particular, the geographic part of the hierarchy (once fully elaborated) is similar to geographic thesauri such as Getty's thesaurus [26]. The graph structure of the hierarchy in Level 1 and some of the motivations in Section 5.2 indicate a neighbourhood to knowledge bases such as WordNet [21, 2], perhaps also to conceptual lattices [49, 23]. From the temporal information encoded in the resource we obtain a direct line to interval-based temporal reasoning [5] and calendar systems [37]. The idea of reasoning based on taxonomies and classification schemes, which plays an important role for the given hierarchy, has been discussed in [20]. One important application area of the resource is the semantic web [52]. Here suitable deduction mechanisms built upon the resource could yield a valuable addition to approaches based on ontologies and description logics [32, 6, 3, 46]. Eventually, our resource shares some ideas with Topic Maps [50]. Topic Maps describe distinct entities and thematic fields using typed relations called "associations", and add typed links to "occurrences" (CVs, home pages, etc.) that yield further information on the topics. Associations are similar to the relations in our Level 2, occurrences are close to the URLs that we use in the formal description of events and other entities. Topics Maps do not use identifiers that "describe" the nature of topics and entities. Hence there is no internal ordering between topics derived from identifiers, as in our approach. Furthermore, no special emphasis is given to temporal and geographic data.

# Bibliography

[1] DIN 1463: Erstellung und Weiterentwicklung von Thesauri. Deutsches Institut für Normung, 1987.

[2] EuroWordNet Consortium. http://www.hum.uva.nl/ ewn/index.html♯2.

[3] ECAI'98 Workshop on Applications of Ontologies and Problem Solving Methods, 1998. http://delicias.dia.fi.upm.es/WORKSHOP/ECAI98/papers.html.

[4] The ACM Computing Classification System, 2001. http://www.acm.org/class/1998/homepage.html.

[5] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[6] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications.* Cambridge University Press, 2003.

[7] 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.

[8] 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.

[9] 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.

[10] Claudio Bettini, Sushil Jajodia, and Sean X. Wang. *Time Granularities in Databases, Data Mining and Temporal Reasoning.* Springer Verlag, 2000.

[11] 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.

[12] 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.

[13] Diana R. Cukierman. *A Formalization of Structured Temporal Objects and Repetition.* PhD thesis, Simon Franser University, Vancouver, Canada, 2003.

[14] Diana R. Cukierman and James P. Delgrande. Expressing time intervals and repetition within a formalization of calendars. *Computational Intelligence*, 14(4):563–597, 1998.

[15] 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.

[16] Nachum Dershowitz and Edward M. Reingold. *Calendrical Calculations.* Cambridge University Press, 1997.

[17] Didier Dubois and Henri Prade, editors. *Fundamentals of Fuzzy Sets.* Kluwer Academic Publisher, 2000.

[18] 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.

[19] 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.

[20] Andrew Fall. *Reasoning with Taxonomies.* PhD thesis, School of Computing Science, Simon Fraser University, 1996.

[21] Christiane Fellbaum, editor. *WordNet - An Electronic Lexical Database.* The MIT Press, May 1998.

[22] D. M. Gabbay, I. Hodkinson, and M Reynolds, editors. *Temporal Logic: Mathematical Foundations and Computational Aspects*, volume 1. Oxford: Clarendon Press, 1994.

[23] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis - Mathematical Foundations.* Springer Verlag, Berlin Heidelberg, 1999.

[24] Jacob E. Goodman and Joseph O'Rourke, editors. *Handbook of Discrete and Computational Geometry.* CRC Press, 1997.

[25] 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.

[26] Getty Research Institute GRI. Getty Thesaurus of Geographic Names On Line. http://www.getty.edu/research/tools/vocabulary/tgn/.

[27] G. Gross and F. Guenthner. Traitement automatique des domaines. *Révue francaise de linguistique appliquée*, 1998.

[28] International Press Telecommunications Council IPTC. IPTC Subject Reference System. http://www.iptc.org/site/subject-codes/.

[29] 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.

[30] 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.

[31] B. Leban, D. D. McDonald, and D. R. Forster. A representation of collections of temporal intervals. In *Proc. of AAAI'86*, pages 367–371, 1986.

[32] Deborah L. McGuiness, Richard Fikes, James Hendler, and Lynn Andrea Stein. DAML+OIL: An Ontology Language for the Semantic Web. *IEEE Intelligent Systems*, 17(5):72–80, 2002.

[33] 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.

[34] 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.

[35] 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.

[36] National Library of Medicine NLM. Unified Medical Language System UMLS. http://www.nlm.nih.gov/research/umls/.

[37] 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.

[38] 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.

[39] Hans Jürgen Ohlbach. Calendrical calculations with time partitionings and fuzzy time intervals. In H. J. Ohlbach and S. Schaffert, editors, *Proc. of PPSWR04*, number 3208 in LNCS. Springer Verlag, 2004.

[40] Hans Jürgen Ohlbach. Fuzzy time intervals and relations – the FuTIRe library. Technical report, Inst. f. Informatik, LMU München, 2004. See http://www.pms.informatik.uni-muenchen.de/mitarbeiter/ohlbach/systems/FuTIRe.

[41] Hans Jürgen Ohlbach. Relations between fuzzy time intervals. In C. Combi and G. Ligozat, editors, *Proc. of the 11th International Symposium on Temporal Representation and Reasoning*, pages 44–51, Los Alamitos, California, 2004. IEEE.

[42] Hans Jürgen Ohlbach. The role of labelled partitionings for modelling periodic temporal notions. In C. Combi and G. Ligozat, editors, *Proc. of the 11th International Symposium on Temporal Representation and Reasoning*, pages 60–63, Los Alamitos, California, 2004. IEEE.

[43] Hans Jürgen Ohlbach. The role of labelled partitionings for modelling periodic temporal notions. httpd://www.informatik.uni-muenchen.de/mitarbeiter/ohlbach/homepage/publications/PRP/abstracts.shtml, 2004. To be published.

[44] Hans Jürgen Ohlbach and Dov Gabbay. Calendar logic. *Journal of Applied Non-Classical Logics*, 8(4), 1998.

[45] Joseph O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1998.

[46] Stefan Schlobach. Description logics and knowledge discovery of data. In H. J. Ohlbach, U. Endriss, O. Rodrigues, and S. Schlobach, editors, *Proceedings of the Seventh Workshop on Automated Reasoning, Bridging the Gap between Theory and Practice*, volume 32 of *CEUR Workshop Proceedings*, July 2000. On-line proceedings are available at http://SunSITE.Informatik.RWTH-Aachen.DE/Publications/CEUR-WS/Vol-32/.

[47] Klaus U. Schulz and Felix Weigel. Systematics and architecture for a resource representing knowledge about named entities. In Jan Maluszynski Francois Bry, Nicola Henze, editor, *Principles and Practice of Semantic Web Reasoning*, pages 189–208, Berlin, 2003. Springer-Verlag.

[48] 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.

[49] Gerd Stumme and Rudolf Wille. *Begriffliche Wissensverarbeitung.* Springer Verlag, Berlin Heidelberg, 2000.

[50] TopicMaps.Org. XML Topic Maps XTM.
http://www.topicmaps.org/xtm/1.0/.

[51] Universal Decimal Classification Consortium UDC. Universal Decimal Classification. http://www.udcc.org/outline/outline.htm/.

[52] W3C. Semantic Web. http://www.w3.org/2001/sw/.

[53] L. A. Zadeh. Fuzzy sets. *Information & Control*, 8:338–353, 1965.