Relations Between Fuzzy Time Intervals

Hans Jürgen Ohlbach
Institut für Informatik, Universität München
email: ohlbach@lmu.de

June 17, 2005

Abstract

This paper serves two purposes. The first purpose is to introduce a new approach for defining point–interval and interval–interval relations for fuzzy time intervals. The basic idea for the interval–interval relations is to extend corresponding point–interval relations to interval–interval relations by averaging (integrating) the point–interval relation over the second interval. The new approach is compared with an existing approach by Nagypál and Motik.

The second purpose is to show how these relations can be easily defined with the GeTS language (GeTS stands for GeoTemporal Specifications). This is a typed functional language with a large number of built–in data types and functions for manipulating temporal notions. The definitions of the point–interval and interval–interval relations are therefore given directly in the GeTS language.

Contents

1 Motivation and Introduction

Time points and time intervals are the basic concepts in many formalisations of temporal notions. In order to make basic concepts useful for practical applications, one has to define operations on them and relations between them. If we take, for example, two time points t_1 and t_2 and a linear ordering of the time structure, which we assume throughout this paper, there are the basic relations $t_1 < t_2$, $t_1 = t_2$ and $t_1 > t_2$. The relations between a time point t and a (crisp) time interval I are slightly more complex. First of all, one needs to distinguish whether I is convex or not. If it is not convex it consists of several unconnected subintervals. Secondly, one can consider or ignore the underlying metric of the time axis. If the metric is ignored and I is convex there are the usual five point–interval relations:

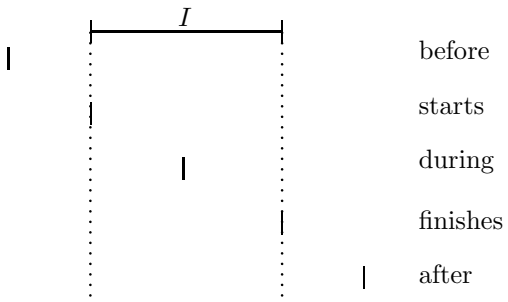


Figure 1: Point–Interval Relations

These relations are well defined even if I is infinite, or if one distinguishes whether I is closed or open at one or both sides. If I is open at the left side then t *starts* I may be true even if t*during* I is false. For a non-convex interval I there are some more relations: 't is within the n'the component of I', 't is in a gap of I', 't is within the n'th gap of I' etc.

The metric of the time axis gives rise to more relations. With a metric one can measure the location and length of the interval. Therefore, at least for finite intervals I, it makes sense to define 't is in the first half of I', or more general, 't is in the n'th m'th of I'. These relations can also be obtained in an indirect way by cutting out the first half, or, more general, the n'th m'th of I, and evaluating point–interval relations between t and the cut out part of I. This way one can easily define relations like 't is before (after, ...) the n'th m'th of I'.

In the next stage we can consider interval–interval relations. Allen's interval–interval relations [1] are the basic relations between two crisp intervals (without metric).

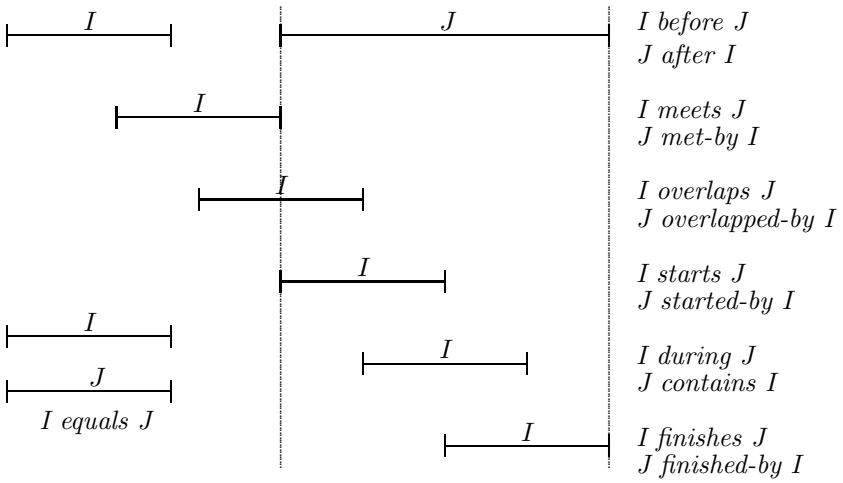


Figure 2: Interval–Interval Relations

Natural language has a lot more interval–interval relations They, however, usually rely on the time metric. 'is long before' or 'is close to' are examples.

Things get a lot more complicated if we consider fuzzy time intervals instead of crisp time intervals. Fuzzy time intervals can be used to represent fuzzy notions like 'around noon', 'late night', 'during sunrise'

etc. If the intervals are in fact fuzzy, one expects that the relations 'before', 'during' etc. are also no longer simple relations with a Boolean result, but binary functions with a fuzzy value as result. Unfortunately, there are no unique and natural ways to generalise the relations between crisp intervals to relations between fuzzy intervals. There are many different possibilities, and it depends on the applications, which one is the most appropriate one. A tool for representing and manipulating fuzzy intervals should therefore provide several of these possibilities, or, even better, allow the user to define his favourite version of the relations.

The purpose of this paper is therefore twofold:

1. a new approach for interval–interval relations between fuzzy time intervals is presented. The idea of this new approach is to take a suitable point–interval relation and extend it to an interval–interval relation by averaging the point–interval relation over the second interval. The details of this approach are presented and compared to other approaches;

2. the fuzzy interval–interval relations are used as a case study for a part of the CTTN system (Computational Treatment of Temporal Notions) [5]. The CTTN system provides data structures and operations for various basic temporal notions. In particular it contains time points, crisp and fuzzy time intervals [6], labelled partitionings for representing periodic temporal notions and calendar systems [8]. In addition it has the GeTS (GeoTemporal Specification) language for specifying and computing with application specific temporal notions. In this paper it is shown how the GeTS language can be used to define crisp and fuzzy point–interval and interval–interval relations.

No argument is given in this paper that a particular version of the relations is good for a particular application. Instead, we want to give evidence that the CTTN system and in particular the GeTS language is good for specifying even very complex relations in a simple and intuitive way.

2 Time Points and Time Intervals

In this section we briefly introduce the basic mathematical concepts of time points and fuzzy temporal intervals as well as their GeTS encoding. More details can be found in [6] and [7].

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 is sufficient to restrict the representation of concrete time points to *integers*. The GeTS language has a type `Time`, which internally uses 64 bit integer for modelling time points, together with a special representation of negative and positive infinity.

Definition 2.1 (GeTS: `isInfinity`) *The functions*

`isInfinity`(*time*) *of type* `Time->Bool` (1)

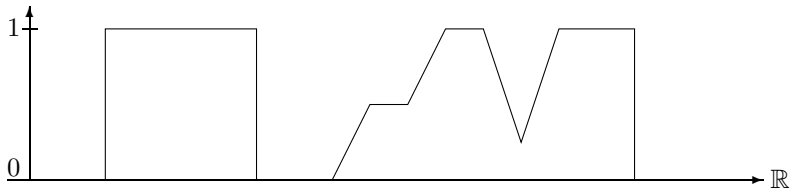
`isInfinity`(*time*, `positive/negative`) *of type* `Time*PosNeg->Bool` (2)

can be used to check whether the time point time represents an infinity (version `(1)`*).* `isInfinity`(*time*, `positive`) *checks whether 'time' represents* $+\infty$ *and* `isInfinity`(*time*, `negative`) *checks whether 'time' represents* $-\infty$*.* ∎

The next important data type are time intervals. Fuzzy intervals are usually defined through their membership functions [9, 2]. A membership function maps a base set, in our case $\mathbb{R}$, to real numbers between 0 and 1.

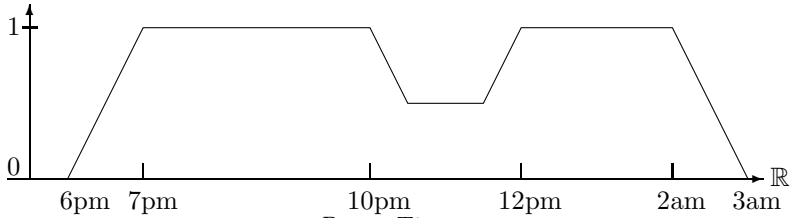
Definition 2.2 (Fuzzy Time Intervals) *A* fuzzy membership function *in CTTN is a total function* $f : \mathbb{R} \mapsto [0, 1]$ *which does not need to be continuous, but it must be integratable. The* fuzzy interval I_f *that corresponds to a fuzzy membership function* f *is* $I_f \stackrel{\text{def}}{=} \{(t, y) \subseteq \mathbb{R} \times [0, 1] \mid y \leq f(t)\}$*. Given a fuzzy interval* I *we usually write* $I(t)$ *to indicate the corresponding membership function.* ∎

This definition comprises single or multiple crisp or fuzzy intervals like these:



Crisp and Fuzzy Intervals

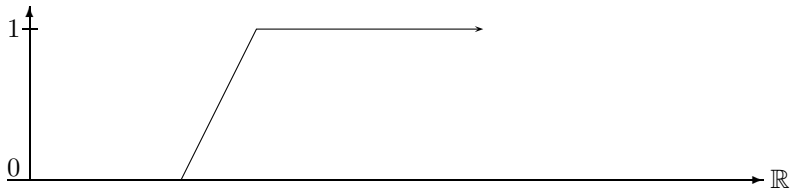
It also comprises finite fuzzy intervals like this one:



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.

The fuzzy intervals can also be infinite. For example, the term 'after tonight' may be represented as 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'

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. Fuzzy time intervals with finite kernel are of particular interest because, although they may be infinite, they can easily be implemented with finite data structures. Therefore CTTN represents only fuzzy intervals with finite kernel. The 'maxRegion' of the interval I is the interval $[t_1, t_2[$ where t_1 is the leftmost time point t where $I(t)$ is maximal and t_2 is the rightmost such value.

Definition 2.3 (Support, Core, Kernel, MaxRegion) *Let I be a fuzzy time interval. Let $y_1 \stackrel{\text{def}}{=} lim_{t \mapsto -\infty} I(t)$ and $y_2 \stackrel{\text{def}}{=} lim_{t \mapsto +\infty} I(t)$. Let I_{max} the maximal fuzzy value of I.*

We define

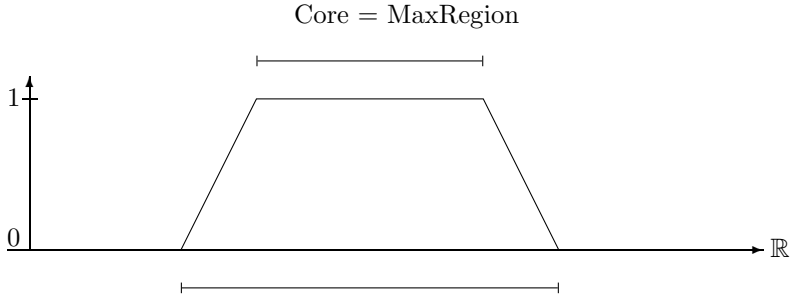
$S(I) \stackrel{\text{def}}{=} \{t \mid I(t) \neq 0\}$ (*support*)

$C(I) \stackrel{\text{def}}{=} \{t \mid I(t) = 1\}$ (*core*)

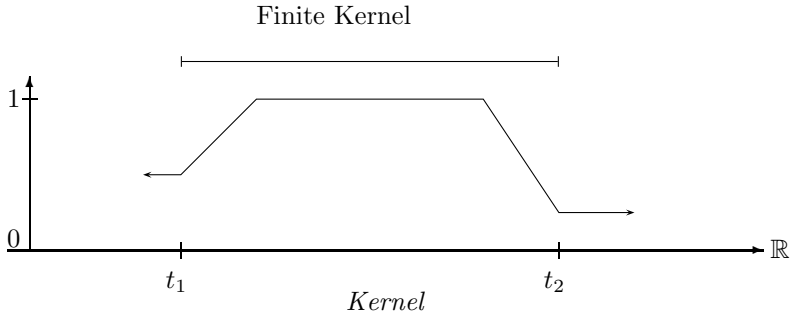
$K(I) \stackrel{\text{def}}{=} [\sup\{t \mid \forall s < t : I(s) = y_1\}, \inf\{t \mid \forall s > t : I(t) = y_2\}[$ (*kernel*)

$M(I) \stackrel{\text{def}}{=} [sup\{t \mid \forall s < t : I(s) < I_{max}\}, \inf\{t \mid \forall s > t : I(t) < I_{max}\}[$ (*maxRegion*)

■



Support, Core and MaxRegion



Kernel

The GeTS language has some constructs which allow one to access the different regions of an interval.

Definition 2.4 (GeTS: `Region`, `Side`, `hull` and `point`)
`Region` *is an enumeration type with values* `support`, `core`, `kernel`, `maximum`.
`Side` *is an enumeration type with values* `left` *and* `right`.
The function `hull`*(I,region) of type* `Interval * Region -> Interval` *computes the corresponding region of the interval* I*, i.e.* $S(I)$, $C(I)$, $K(I)$ *or* $M(I)$*, as a, possibly non-convex, crisp interval.*
The function `point`*(I,side,region) of type* `Interval * Side * Region -> Time` *accesses the left/right end of the corresponding region of the interval* I*.* ■

The `sup` and `inf` functions in GeTS compute the supremum/infimum of the fuzzy values of an interval. They will be frequently used in the definitions of the point–interval and interval–interval relations.

Definition 2.5 (GeTS: `sup` and `inf`) *Let* I *be a fuzzy time interval.*

$$\texttt{sup}(I) \stackrel{\text{def}}{=} \sup\{I(t) \mid t \in \mathbb{R}\} \quad \textit{supremum of fuzzy values}$$
$$\texttt{inf}(I) \stackrel{\text{def}}{=} \inf\{I(t) \mid t \in \mathbb{R}\} \quad \textit{infimum of fuzzy values}$$

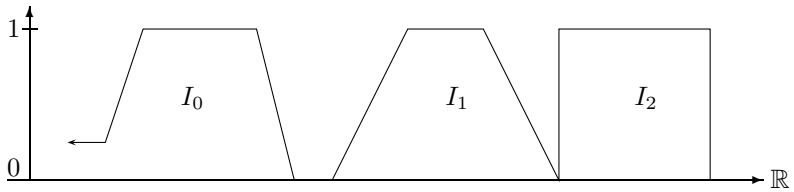
■

Tow useful predicates on intervals are `isEmpty` and `isCrisp`.

Definition 2.6 (GeTS: `isEmpty` and `isCrisp`) *Let* I *be a fuzzy time interval.*
The function `isEmpty`(I) *checks whether the interval* I *is empty.*
The function `isCrisp`(I) *checks whether the interval* I *is crisp (possibly non-convex).*
The function `isCrisp`$(I, \texttt{left})$ *checks whether the interval* I *is crisp at its left side.*
The function `isCrisp`$(I, \texttt{right})$ *checks whether the interval* I *is crisp at its right side.* ■

Components
Fuzzy time intervals can consist of several different components. A component is a sub-interval of a fuzzy interval such that the left and right end is either the infinity, or the membership function drops down to 0.

Example:



Components

Definition 2.7 (GeTS: `component`)
The function `component(I,n)` *of type* `Interval * Integer -> Interval` *extracts the n'th component of the interval* I*.* ■

2.1 Set Operations on Intervals

There are no uniquely defined set operations $\cap$ (intersection), $\cup$ (union) and $\setminus$ (set difference) for fuzzy intervals. Instead there are axiomatically characterised classes of unary and binary functions on fuzzy intervals which can take over the role of the standard set operations [2]. The GeTS language has constructs for defining these functions. Some of the more natural and more frequently used set operations, however, are built-in.

For the purposes of this document, it is sufficient to know the simplest ones.

Definition 2.8 (GeTS: Set Operations) *Let I and J be two fuzzy intervals and t a time point. The following GeTS set operators are available, among others:*

$$\begin{aligned}
&\texttt{complement}(I)(t) &&\stackrel{\text{def}}{=} 1 - I(t)\\
&\texttt{union}(I,J)(t) &&\stackrel{\text{def}}{=} max(I(t),J(t))\\
&\texttt{intersection}(I,J)(t) &&\stackrel{\text{def}}{=} min(I(t),J(t))\\
&\texttt{setdifference}(I,J) &&\stackrel{\text{def}}{=} intersection(I, complement(J))
\end{aligned}$$

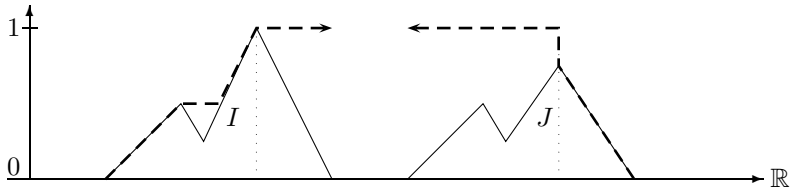
■

2.2 Basic Operations on Fuzzy Intervals

In this section we list some of the built-ins of the GeTS language which are used in the definitions of the point–interval and interval–interval relations.

2.2.1 Extend

`extend`$(I,$`positive`$)$ follows the left part of the monotone hull of the interval until the left maximum is reached and then stays at fuzzy value 1. `extend`$(I,$`negative`$)$ is the symmetric variant of `extend`$(I,$`positive`$)$.

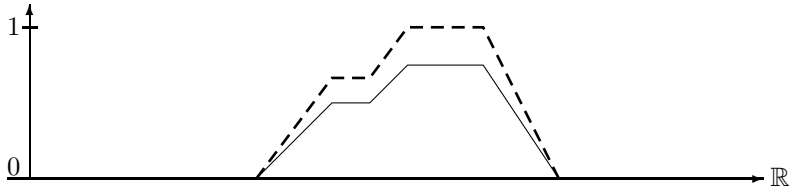


`extend(I,positive)` *and* `extend(J,negative)`

`extend`$(I,$`positive`$)$ is useful for implementing a *before*–relation because only the left part of I is relevant for evaluating *before*. `extend`$(I,$`negative`$)$, on the other hand, can be used for an *after*–relation.

2.2.2 Scaleup

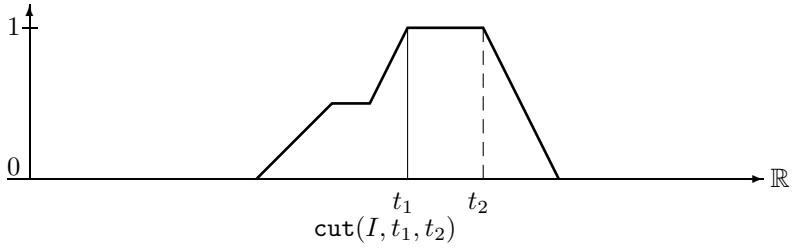
The `scaleup`-function is different to the *identity* function only if `sup`$(I) \neq 1$. In this case it scales the membership function up such that its maximum value is 1.



`scaleup`

2.2.3 Cut

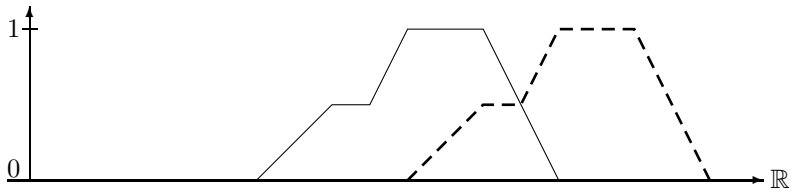
`cut`(I,t_1,t_2) just cuts the piece between t_1 and t_2 out of the interval I. The resulting interval is closed at t_1 and half open at t_2.



`cut`(I,t_1,t_2)

2.2.4 Shift

`shift`(I,n) just moves the interval I by n time units.

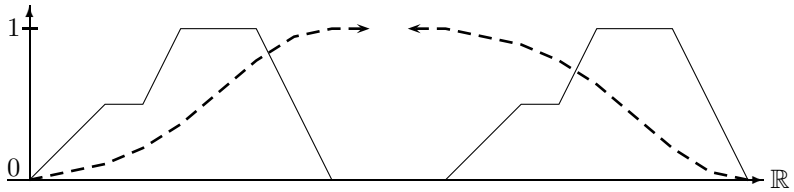


$\texttt{shift}(I, 20)$

2.2.5 Integrate

This operator integrates over the membership function and normalises the integral to values ≤ 1. If the fuzzy interval I is finite then

$$\texttt{integrate}(I, \texttt{positive})(t) = \frac{\int_{-\infty}^{t} I(y)dy}{|I|} \qquad \text{and} \qquad \texttt{integrate}(I, \texttt{negative})(t) = \frac{\int_{t}^{+\infty} I(y)dy}{|I|}.$$



$\texttt{integrate}(I, \texttt{positive})$ *and* $\texttt{integrate}(I, \texttt{negative})$

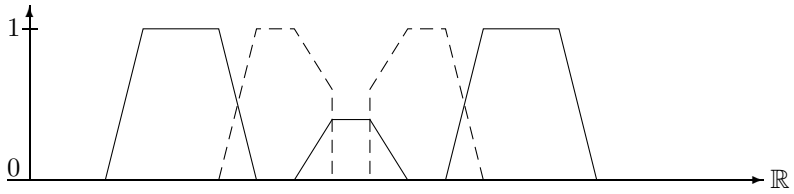
The integration operator for infinite intervals I with finite kernel turns the interval into a constant function which does no longer depend on the finite part of I.

If the infinite fuzzy interval I has a finite kernel with $I_1 \stackrel{\text{def}}{=} I(-\infty)$ and $I_2 \stackrel{\text{def}}{=} I(+\infty)$ then

$$\texttt{integrate}(I, \texttt{positive})(t) = \frac{I_1}{I_1 + I_2} \qquad \text{and} \qquad \texttt{integrate}(I, \texttt{negative})(t) = \frac{I_2}{I_1 + I_2}.$$

2.2.6 Invert

The `invert` function is almost like the standard negation function, except that $\texttt{invert}(I)$ 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 $\texttt{invert}(I)$ drops down to 0 between the first and last maximum of the middle component.



$\texttt{invert}(I)$

2.2.7 Fuzzification

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

Definition 2.9 (Fuzzification) *There are three different versions of the* `fuzzify` *function in GeTS. The first version allows one to specify the part of the interval I which is to be fuzzified in terms of*

percents of the interval length. The second version needs absolute coordinates. The third version fuzzifies both sides of the interval.

1. `fuzzify`$(I, $`linear/gaussian,left/right`$, \mathit{increase}, \mathit{offset})$
   of type
   `Interval`, `Fuzzify`, `Side`, `Float`, `Float` $\mapsto$ `Interval`
2. `fuzzify`$(I, $`linear/gaussian,left/right`$, x1, x2, \mathit{offset})$
   of type
   `Interval`, `Fuzzify`, `Side`, `Time`, `Time`, `Time` $\mapsto$ `Interval`
3. `fuzzify`$(I, $`linear/gaussian`$, \mathit{increase}, \mathit{offset})$
   of type
   `Interval`, `Fuzzify`, `Float`, `Float` $\mapsto$ `Interval`

■

The second parameter determines whether a linear or Gaussian increase is to be imposed on the interval. The third parameter determines whether the increase is from left to right or from right to left. *increase* is a Float number in percent. *increase* = 10 means that the region to be modified consists of the first/last 10% of the *kernel* of the interval. *offset* is also a float number in percent. *offset* = 20 means that the interval is to be widened by 20% of the *kernel* of the interval. To this end the fuzzified part of the interval is shifted back (second parameter = `left`) or forth (second parameter = `right`) 20% of the kernel size.

$x1$ and $x2$ in the second version of the fuzzify function allows one to determine the part of the interval to be fuzzified in absolute coordinates. `fuzzify`$([0, 100], $`linear`$, $`left`$, 20, 70, 0)$, for example, yields a polygon [(20,0) (70,1) (100,1) (100,0)]. `fuzzify`$([0, 100], $`linear`$, $`right`$, 20, 70, 0)$, on the other hand, yields a polygon [(0,0) (0,1) (20,1) (70,0)]. The offset widens the polygon: `fuzzify`$([0, 100], $`linear`$, $`right`$, 20, 70,$ 20), yields [(0,0) (0,1) (60,1) (90,0)].

Finally, the third version is similar to the first version. The difference is that both sides of the interval are fuzzified.

3 Point–Interval Relations

The five basic relations between a time point and a *crisp* interval I are *before*, *starts*, *during*, *finishes* and *after* (cf. Fig. 1). If the intervals possess a metric, which is the case for time intervals over the real numbers, there are infinitely many more point–interval relations. Examples are 'during the first half' or 'in the middle of the third quarter'.

If the metric is given in terms of time units of a calendar system there are even more complex point–interval relations. 'before the year 2006', for example, can mean 'some weeks or months before 2006', but not '1000 years before 2006'. Linguists analyse the precise meaning of such expressions. The GeTS language provides constructs for defining these more complex relations, but this is not subject of this paper.

For non-convex intervals there are even more point–interval relations, for example 'between the components' or 'between the second and third component' etc.

A point–interval relation R can also be represented as a function which maps an interval to an interval. For example, the 'before' relation can be represented as the function which maps an interval I to the interval J containing all the points before I. This formalisation of point–interval relations as functions of type `Interval` $\mapsto$ `Interval` can now be generalised to fuzzy intervals in a way that fuzzy point–interval relations yield fuzzy values instead of Boolean values.

Definition 3.1 (Point–Interval Relations as Functions)

1. *A fuzzy point–interval relation $R(t, I)$ is a function that maps a time point t and an interval I to a fuzzy value.*
2. *If I is a fuzzy interval and R' is a function of type* `Interval` $\mapsto$ `Interval` *then R with the definition:*

$$R(t, I) \stackrel{\text{def}}{=} R'(I)(t)$$

is the corresponding fuzzy point–interval relation. ■

With this definition one can turn any `Interval` $\mapsto$ `Interval` function into a fuzzy point–interval relation. Since there are infinitely many of them, it is by no means obvious how to characterise some of them as, for example, fuzzy 'before' relations, or fuzzy 'starts' relations etc. One could, for example,

require that an `Interval` $\mapsto$ `Interval` function B represents a fuzzy 'before' relation if in the limit case where I is a crisp interval, $B(I)$ is the ordinary crisp 'before' function. This, however, is a property which is not always desired. For example, it should be possible to assign a non-zero fuzzy value to the statement 'the movie ends *before* the concert', even if the movie ends one minute after the concert starts.

In fact, there is no clear mathematical characterisation of `Interval` $\mapsto$ `Interval` functions which allows one to distinguish the corresponding different kinds of point–interval relations. Now, what can we do? A pragmatic approach is to develop tools which allows the user to specify his favourite version of fuzzy point–interval relations in an easy and intuitive way. In order to make the tool more useful, it should contain some predefined versions of the different point–interval relations, which are good enough for most applications.

Three Versions of the Point–Interval Relations

For each of the standard point–interval relations, *before*, *starts*, *during*, *finishes* and *after*, we provide three different versions. All three versions are functional, i.e. they return an interval as a result.

Version 1:
The first version is essentially a pure crisp version of the relation. The type of the function is `Interval` * `Region` $\mapsto$ `Interval`. `Region` is one of the key words `support`, `core`, `kernel`, `maximum`. This version takes the corresponding crisp region $S(I)$, $C(I)$, $K(I)$ or $M(I)$ of the interval I (Def. 2.3) and computes the crisp relation.

Version 2:
This is the most general version. It has some extra parameters which are functions for manipulating the interval in a certain way.

Version 3:
Version 3 is a specialisation of version 2. The extra parameters are instantiated with suitable concrete functions. This version is the 'standard' fuzzy point–interval relation.

3.1 Point–Interval 'Before' and 'After' Relations

The *before* function maps an interval I to an interval containing all the points before I.

Definition 3.2 (Point–Interval 'Before' Functions)

```
1. PIR::Before(Interval I, Region R) =
        if isEmpty(I) then []
        else Let t=point(I,left,R) in
                 if(isInfinity(t)) then [] else [(t,1.0),(t,0.0)]

2. PIR::Before(Interval I, Interval->Interval C, Interval->Interval E) =
        if isEmpty(I) then []
        else C(E(I))

3. PIR::Before(Interval I) =
        if isEmpty(I) then []
        else complement(extend(I,positive))
```

■

Version 1:
This version turns the interval into a crisp interval containing the points before the corresponding region of the interval.

The expression `Let t= point(I,left,R)` causes that this value is bound to the local variable `t`. `[(t,1.0),(t,0.0)]` constructs a crisp interval $]-\infty, t[$.

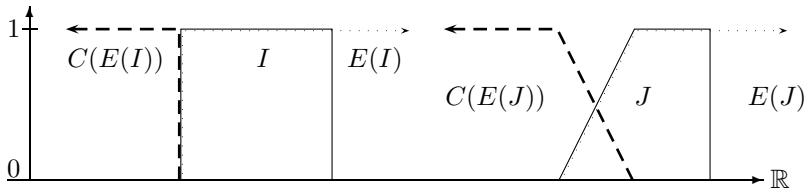
Version 2:
This is the most general version of a 'before' function. Both extra parameters `C` and `E` are in principle arbitrary (total) `Interval` $\mapsto$ `Interval` functions. The idea behind this function is that the parameter `E` is a function which isolates the front part of the interval and extends it to $+\infty$. The parameter `C` must be a complement function which complements the front part.

Version 3:
is an instance of version 2 where `E` is the function `lambda(I) extend(I,positive)` and `C` is the standard complement function: `complement`$(I)(t) = 1 - I(t)$ (Def. 2.8). `PIR::Before(Interval I)` is actually a shortcut for

```
PIR::Before(I,lambda(Interval J) complement(J),
              lambda(Interval J) extend(J,positive))
```

We illustrate version 3 and version 2 with a few examples.

Example 3.3 (`PIR::Before`) $E(I) \stackrel{\text{def}}{=} $`extend(`$I$`,positive)` *and* $C(I) \stackrel{\text{def}}{=} $`complement(`$I$`)`.



`PIR::Before`

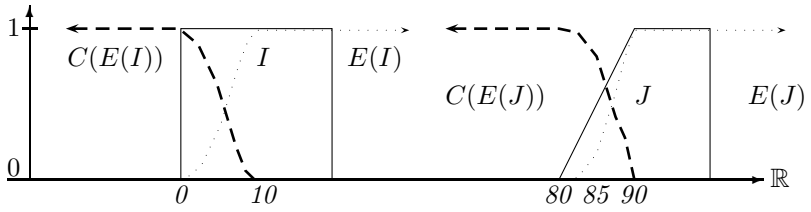
■

The left picture shows `PIR::Before(I)` where `I` is a crisp interval. This yields the ordinary crisp 'before' relation. `PIR::Before(I)` yields the same result as version 1, `PIR::Before(I,R)` where `R` is any of the admissible key words.

The next examples show how the *before*–relation can be made fuzzy, even for crisp intervals. The pictures are produced with the following call of the `PIR::Before` relation.

```
PIR::Before(I,lambda(J) complement(J),
              lambda(J) extend(fuzzify(I,gaussian,left,50,0),positive))
```

Example 3.4 (Gaussian `PIR::Before`) $E(I) \stackrel{\text{def}}{=} $`extend(fuzzify(`$I$`,gaussian,left,50,0),positive)` *and* $C(I) \stackrel{\text{def}}{=} $`complement(`$I$`)`.



`PIR::Before` *with Gaussian Fuzzification Function*

■

The expression `fuzzify(`I`,gaussian,left,50,0)` (Def. 2.9) causes that the front 50% of I is multiplied with a gaussian function. `extend(fuzzify(`I`,gaussian,left,50,0),positive)` then yields the front part of the fuzzified I, extended to the infinity.

Notice that `fuzzify(extend(`I`,positive),gaussian,left,50,0)` does not work. The reason is that `extend(`I`,positive)` produces an infinite interval, and 50% of an infinite interval is not defined.

'After':
The point–interval 'after' functions are very analogous to the corresponding 'before' functions. In particular, the general version 2 has exactly the same definition as version 2 of PIR::Before. In order to get an acceptable result of version 2 of PIR::After, one has to pass a function like

`lambda(Interval` J`) extend(`J`,negative)`

as the `E` parameter to PIR::After. `extend(`J`,negative)` extracts the back part of J and extends it to $-\infty$.

Definition 3.5 (Point–Interval 'After' Functions)

```
1. PIR::After(Interval I, Region R) =
        if isEmpty(I) then []
        else Let t = point(I,right,R) in
                if(isInfinity(t)) then [] else [(t,0.0),(t,1.0)]

2. PIR::After(Interval I, Interval->Interval C, Interval->Interval E) =
        if isEmpty(I) then []
        else C(E(I))

3. PIR::After(Interval I) =
        if isEmpty(I) then []
        else complement(extend(I,negative))
```

■

3.2 Point–Interval 'Starts' and 'Finishes' Relations

The standard *starts*-relation t *starts* I for crisp intervals I yields 1 if t is the starting point of I, and 0 everywhere else. This corresponds to an almost empty fuzzy set with a single peak right at the start of I. The generalisation of this simple definition to fuzzy intervals is version 1 in Def. 3.6 below. `t = point(I,left,R)`, where `R` is again one of the key words `support`, `core`, `kernel` or `maximum`, determines the left boundary of the corresponding region as the start of the interval. The generated interval is `[t,t+1]`, which internally is mapped to the half open interval $[t, t+1[$. Since it is half open, its fuzzy value is 1 for the time point t and 0 for the time point $t+1$, as expected.

Definition 3.6 (Starts)

```
1.  PIR::Starts(Interval I, Region R) =
        if isEmpty(I) then []
        else Let t=point(I,left,R) in
                if(isInfinity(t)) then [] else [t,t+1]

2. PIR::Starts(Interval I, Interval->Interval E, Interval->Interval B,
        (Interval*Interval)->Interval Intersect) =
        case isEmpty(I) or isInfinite(I,left): [],
             isCrisp(I,left): scaleup(Intersect(E(I),shift(B(I),1)))
        else                  scaleup(Intersect(E(I),B(I)))

3. PIR::Starts(Interval I) =
        case isEmpty(I) or isInfinite(I,left): [],
             isCrisp(I,left): Let t=point(I,left,support) in [t,t+1]
        else scaleup(intersection(extend(I,positive),PIR::Before(I)))
```

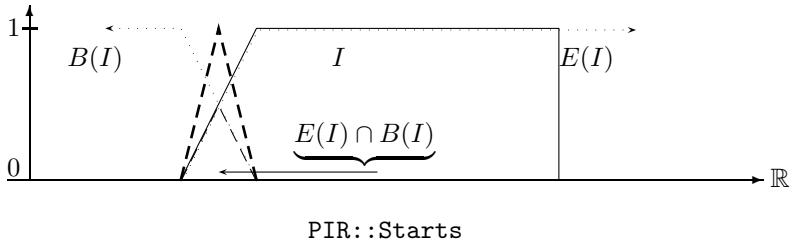
■

Version 1:
This version yields a single peak even if the front part of the interval rises smoothly.

Version 2:
The more general version 2 is more fuzzy in this case. Version 2 of PIR::Starts accepts two extra `Interval -> Interval` functions as arguments. The first one, `E`, is supposed to extract the front part of the interval and extend it to $+\infty$. The second one, `B`, should be one of the point–interval 'before' functions. `scaleup(Intersect(E(`I`),B(`I`)))` intersects `E(`I`)` and `B(`I`)` and scales it up such that the maximum fuzzy value of the intersection is 1.

Version 3:
This version is an instance of version 2 with `E = lambda(Interval` J`) extend(`J`,positive)` and `B = lambda(Interval` J`) PIR::Before(`J`)`. `Intersect` is the standard intersection function (Def. 2.8).

Example 3.7 (Standard `PIR::Starts`-function) *Version 3 of* `PIR::Starts` *generates the following result. The dashed line indicates the scaled up intersection.*



`PIR::Starts`

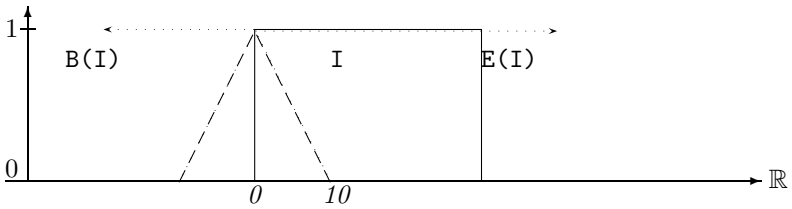
■

There is a small technical problem if we apply `Intersect`$(E(I), B(I))$ to a crisp interval. With the choice of the `E` and `B` functions as in Example 3.7, the intersection of `E(I)` and `B(I)` is empty. The second clause in version 2 of `PIR::Before` takes care of this by shifting `B(I)` one time unit into the future. This way, however, one gets again just a single peak at the start of `I`. If this peak is to be widened in order to get a more fuzzy 'starts' relation even for crisp intervals, there are two possibilities. Either the definition of `PIR::Before` is changed for this case, or you take other `B` and `E` functions. The following definitions

```
E = lambda(Interval J) extend(fuzzify(J,left,linear,30,0),positive) and
B = lambda(Interval J) complement(extend(fuzzify(J,left,linear,30,10),positive)),
```

for example, work for finite intervals. The next picture shows how they fuzzify the start of the interval.

Example 3.8 (`PIR::starts` with fuzzy `E` and `B`)



`PIR::starts`

■

'Finishes':
The definition of the functions `PIR::Finishes` below are analogous to the definition of the `PIR::Starts` functions. The difference is that the `E` function is expected to extract the end of the interval and extend it to $-\infty$. Instead of the 'before' function B, we need an 'after' function A.

Definition 3.9 (Finishes)

```
1. PIR::Finishes(Interval I, Region R) =
        if isEmpty(I) then []
        else Let t=point(I,right,R) in
                 if(isInfinity(t)) then [] else [t,t+1]

2. PIR::Finishes(Interval I, Interval->Interval E, Interval->Interval A,
        (Interval*Interval)->Interval Intersect) =
        case isEmpty(I) or isInfinite(I,right):  [],
             isCrisp(I,right): scaleup(Intersect(shift(E(I),1),A((I))))
        else                   scaleup(Intersect(E(I),A(I)))

3. PIR::Finishes(Interval I) =
        case isEmpty(I) or isInfinite(I,right):  [],
             isCrisp(I,right):  Let t=point(I,right,support) in [t,t+1]
        else scaleup(intersection(extend(I,negative),PIR::After(I)))
```

■

3.3 Point–Interval 'During' Relations

The simplest version of the 'during' relation is just the identity: t *during* $I \stackrel{\text{def}}{=} I(t)$. This yields 0 for all t outside I and the fuzzy membership value for all t inside I. This is version 4 of `PIR::During` below.

Definition 3.10 (`PIR::During`)

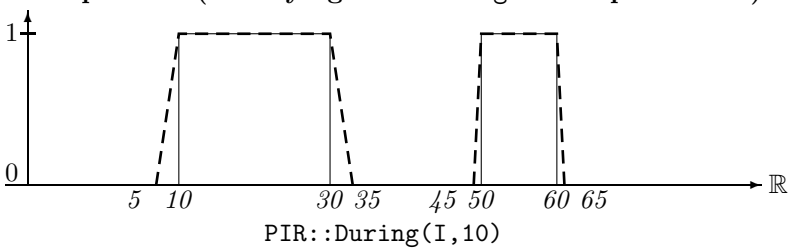
```
1. PIR::During(Interval I, Region R) =  hull(I,R)

2. PIR::During(Interval I, Interval->Interval O,
               (Interval*Interval)->Interval Union) =
        if isEmpty(I) then  []
        else componentwise(I,[],O,Union)

3. PIR::During(Interval I, Float percent) =
        if isEmpty(I) then  []
        else componentwise(I,[],
                lambda(Interval J) fuzzify(J,linear,percent,percent),
                lambda(Interval J, Interval K) union(J,K))

4. PIR::During(Interval I) =  I
```

■

Version 1:
If we insist that `PIR::During` yields a crisp result, we can take version 1. `hull`(I, R) computes $S(I)$, $C(I)$, $K(I)$ or $M(I)$ (Def. 2.3), depending on the parameter R.

Versions 2 and 3:
Versions 2 and 3 of `PIR::During` are more fuzzy versions of *during*. They can return non-zero values even for crisp intervals and time points outside the interval. The basic idea of version 2 is to apply a fuzzification operator O to all components of the interval and to join the fuzzified components with the `Union` operator. Version 3 makes this explicit. It uses the fuzzification operator `fuzzify(`J`,linear,percent,percent)` which fuzzifies the (finite) interval at both ends. The amount of fuzzification is controlled by the `percent` parameter. The next picture shows an example.

Example 3.11 (Fuzzifying `PIR::During` for crisp intervals)



■

3.4 Point–Interval Relations for Non-Convex Intervals

Non-convex crisp intervals consist of several unconnected components. In this case a time point t can also be in any of the gaps between the components. This gives rise to a general 'in the gap' point–interval relation. More specific is a 'in the k'th gap' relation, which determines the particular gap containing t. The same relations can also be defined for non-convex fuzzy intervals. Fuzzy intervals, however, can be non-convex in two ways. They can consist of several components, i.e. their membership function drops down to 0 and rises again. This is a similar kind of non-convexness as for crisp intervals. They can, however, also be non-convex if they consist of a single component, but the membership function drops down and then rises again. The definition of `PIR::InTheGap` below yields nontrivial results only if the interval consists of several components.

Definition 3.12 (In The Gap)

```
1. PIR::InTheGap(Interval I) =
          if isEmpty(I) then []
          else invert(I)

2. PIR::InTheGap(Interval I, Integer k) =
          if isEmpty(I) then []
          else component(invert(I),k)
```

■

The `invert` function in the definitions above inverts the gap regions of the intervals (see Sec. 2.2.6).

3.5 Point–Interval Relations for Intervals with Metric

If the underlying time structure has a metric, it is possible to measure the length of the intervals and to subdivide them into halfs or thirds etc. This gives rise to relations like 'in the first half' or 'in the second third', or, in general, 'in the n'th m'th. The function `PIR::During` below cuts out the n'th m'th of the given interval.

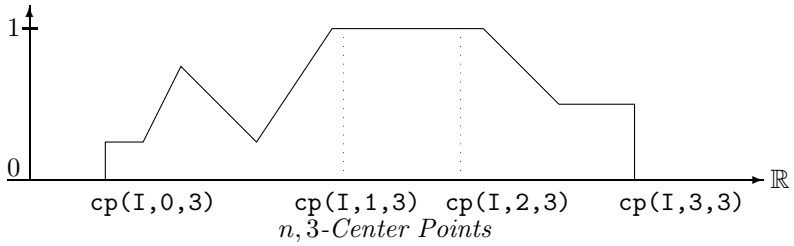
Once a relation like 'in the first half' is defined, it is very natural to define the relation 'in the middle of the first half'. The crisp definition 2 in Def. 3.13 below of this relation has just one single peak in the middle of the first half. The version 3 in Def. 3.13 of `PIR::InTheMiddle` widens this peak by cutting out a slice around this peak. The width of this slice is controlled by the parameter `k`. The larger the `k` the smaller the slice.

Definition 3.13 (Relations with Metric)

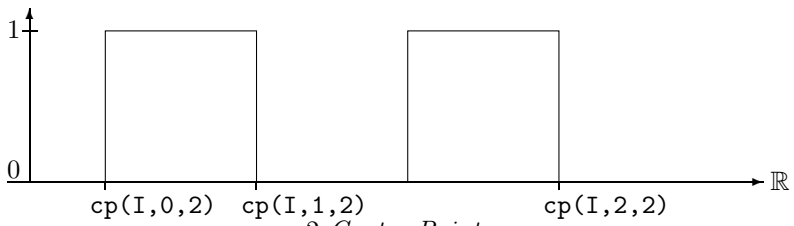
```
1. PIR::During(Interval I, Integer n, Integer m) =
        if(isEmpty(I) or (n < 0) or (m < 0) or (m <= n))  then []
        else cut(I,centerPoint(I,n,m),centerPoint(I,n+1,m))

2. PIR::InTheMiddle(Interval I, Integer n, Integer m) =
        Let t = centerPoint(I,2n+1,2m) in [t,t+1]

3. PIR::InTheMiddle(Interval I, Integer n, Integer m, Integer k) =
        if(isEmpty(I) or (n < 0) or (m < 0) or (m <= n) or (k < 0))  then []
        else Let k1 = pow(2,k) in
             Let n1 = 2*n + 1  in
             Let m1 = k1*2*m   in
                 cut(I,centerPoint(I,k1*n1 - 1,m1),centerPoint(I,k1*n1 + 1,m1))
```

■

Crisp intervals can easily be measured by summing up the length of the components. Fuzzy intervals can be measured by integrating over their membership functions. The function `centerPoint(I,n,m)` in the definitions above computes the `n`,`m` center point of the interval. For example, `centerPoint(I,1,2)` computes the middle point of the interval. If $t =$ `centerPoint(I,1,2)` then the integral over `I` up to t has the same value as the integral over `I` after t.



$n, 3$-*Center Points*



$n, 2$-*Center Points*

Middle Points:
The middle point between the center points `centerPoint(I,n,m)` and `centerPoint(n+1,m)` is just `centerPoint(I,2n+1,2m)`. For example, the middle point in the first half of `I` is `centerPoint(I,1,4)` and the middle point in the second half is `centerPoint(I,3,4)`. This is exploited in the definition of `PIR::InTheMiddle`. The parameter `k` controls the size of the slice around this middle point which is cut out of `I`.

4 Interval–Interval Relations

Allen's seven interval relations [1] are the basic relations between two crisp intervals (cf. Fig. 2). The extension of these relations to fuzzy intervals is neither obvious nor unique. There is one approach proposed by Nagypál and Motik [4], and this paper introduces a second approach.

4.1 Nagypál and Motik's Interval–Interval Relations

The basic idea of Nagypál and Motik's approach is to extend the requirements for the crisp interval–interval relations, which are mostly conditions on the end points of the intervals, to conditions on the starting and finishing sections of the fuzzy intervals. We summarise how these relations work. More details are given in the original paper [4].

Definition 4.1 (Nagypál and Motik's Interval–Interval Relations)
Let I and J be two fuzzy intervals.

$$
\begin{array}{lcl}
\text{before}(I,J) & \stackrel{\text{def}}{=} & sup_t(((1-E^-(I)) \cap (1-E^+(J)))(t)) \\
\text{meets}(I,J) & \stackrel{\text{def}}{=} & min(inf_t((E^-(I) \cup E^+(J))(t)), \\
 & & \quad inf_t(((1-E^-(I)) \cup (1-E^+(J)))(t))) \\
\text{overlaps}(I,J) & \stackrel{\text{def}}{=} & min(sup_t((E^+(I) \cap (1-E^+(J)))(t)), \\
 & & \quad sup_t((E^-(I) \cap E^+(J))(t)), \\
 & & \quad sup_t(((1-E^-(I)) \cap E^-(J))(t))) \\
\text{starts}(I,J) & \stackrel{\text{def}}{=} & min(inf_t(((1-E^+(I)) \cup E^+(J))(t)), \\
 & & \quad inf_t((E^+(I) \cup (1-E^+(J)))(t))), \\
 & & \quad sup_t(((1-E^-(I)) \cap E^-(J))(t))) \\
\text{during}(I,J) & \stackrel{\text{def}}{=} & min(sup_t(((1-E^+(I)) \cap E^+(J))(t)), \\
 & & \quad sup_t(((1-E^-(I)) \cap E^-(J))(t))) \\
\text{finishes}(I,J) & \stackrel{\text{def}}{=} & min(inf_t((E^-(I) \cup (1-E^-(J)))(t))), \\
 & & \quad inf_t(((1-E^-(I)) \cup E^-(J))(t)), \\
 & & \quad sup_t((E^+(I) \cap (1-E^+(J)))(t))) \\
\text{equals}(I,J) & \stackrel{\text{def}}{=} & min(inf_t((E^-(I) \cup (1-E^-(J)))(t))), \\
 & & \quad inf_t(((1-E^-(I)) \cup E^-(J))(t)), \\
 & & \quad inf_t(((1-E^+(I)) \cup E^+(J)))(t)) \\
 & & \quad inf_t((E^+(I) \cup (1-E^+(J)))(t)).
\end{array}
$$

$E^+(I)$ stands for `extend(`I`,positive)` *and $E^-(I)$ stands for* `extend(`I`,negative)`. ∎

The relations give quite intuitive results for finite fuzzy intervals consisting of one component only, and without much internal structure. The definitions work also for infinite intervals, but the results may not be very intuitive. Nagypál and Motik's relations behave on crisp intervals just like Allen's interval–interval relations. This is different to the approach which is presented in this paper.

4.2 Operator Based Interval–Interval Relations

The main requirements for the 'operator based' version of the fuzzy interval–interval relations are:

1. even for two crisp intervals we want a fuzzy value as result. The result should of course be 1 if the classical relation yields true, but it should not necessarily jump to 0 when the classical relation yields false;

2. the relations should work for fuzzy time intervals regardless if they consist of one or more components or if they are finite or infinite.

The basic idea for the operator versions of the relations is very simple: since we have the point–interval relations, we can extend the point to an interval in the relation by integrating over the interval's membership function. Thus, the operator version of the interval–interval relations are essentially averaged

point–interval relations. The functional version of a point–interval relation is the *operator* which is used for the averaging process.

In the rest of this section we present the GeTS definitions of four different versions of the interval–interval relations. Version 1 is the crisp version, where the corresponding crisp region (support, core, kernel or maximum) is the basis for the computation. Version 2 is the most general version where some extra functional parameters are used to compute the relation. Version 3 is an instance of version 2 where default values are used for the extra parameters. Finally, version 4 is Nagypál and Motik's version. Version 1 has a Boolean value as result, whereas all the others have a fuzzy value as result.

Visualisation of the Interval–Interval Relations
A fuzzy interval–interval relation between two concrete fuzzy intervals I and J yields a single fuzzy value as a result. In order to show the effect of a particular definition for such a relation we move the interval I in discrete steps along the time axis and compute for each shift of I the relation to J. The resulting fuzzy values are collected in a new fuzzy interval. The GeTS function `showRelation` below shifts the interval I and collects the points (t, y) in a new interval K. t is the time coordinate of the right end of the kernel of the shifted I and $y = F(I, J)$ is the result of the fuzzy relation F. The figure in Example 4.3 shows the result of `showRelation`. There are many more examples of this kind in the subsequent sections. The right end of the kernel of the shifted interval I is, however, not always used for the time coordinate of the generated interval. In some examples we use the middle of the kernel.

```
showRelation(Interval I, Interval J, (Interval*Interval)->Float F,
             Time distance, Integer steps) =
        Let K = [] in
            while (steps >= 0) {
                  pushBack(K,point(I,right,kernel),F(I,J)),
                  I := shift(I,distance),
                  steps := steps - 1}
            K;
```

4.3 Interval–Interval 'before' Relations

Four different versions of the 'before' relation are presented as GeTS definitions.

Definition 4.2 (Interval–Interval 'before' Relations)

```
1. IIR::Before(Interval I, Interval J, Region R) =
        case isEmpty(I): false,
             isEmpty(J): false
        else (point(I,right,R) <= point(J,left,R))

2. IIR::Before(Interval I, Interval J, Interval->Interval B) =
        case isEmpty(I) or isEmpty(J): 0.0,
             isInfinite(I,right) or isInfinite(J,left): 0.0,
             (point(I,right,support) <= point(J,left,support)): 1.0,
             isInfinite(I,left):
                 Let K = intersection(I,J) in
                     if(isEmpty(K)) then 1.0 else integrateAsymmetric(K,B(J))
        else integrateAsymmetric(I,B(J))

3. IIR::Before(Interval I, Interval J) = IIR::Before(I,J,PIR::Before[Interval])

4. IIR::NMBefore(Interval I, Interval J) =
        case isEmpty(I) or isEmpty(J): 0.0,
             isInfinite(I,right) or isInfinite(J,left): 0.0
        else sup(intersection(complement(extend(I,negative)),
                              complement(extend(J,positive))))
```

■

Version 1

This is Allen's version of the crisp *before*–relation. The parameter `R` (`support`, `core`, `kernel` or `maximum`) determines the crisp part of the intervals which are to be compared.

The `<=` in the expression `point(I,right,R) <= point(J,left,R)` may need an explanation. All intervals are interpreted as half open intervals with integer boundaries. If, for example, $I = [0, 10[$ and $J = [10, 20[$ then 10 belongs to J and not to I. Therefore all points of I, including the 10, are definitely before J. Therefore `<=` is appropriate here.

Version 2 and 3

Version 2 and version 3, as a special case of version 2, are the operator based *before*–relations. The first three cases in version 2 of `IIR::Before` concern trivial cases where the result is obvious. The last two cases are the really interesting ones.

Version 2 for Finite Intervals
The definition for finite intervals I and arbitrary intervals J is:

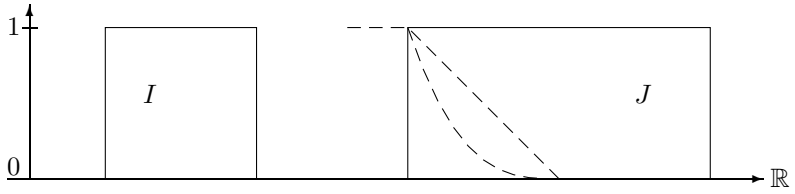
$$\mathit{before}(I, J) \stackrel{\text{def}}{=} \int I(t) \cdot B(J)(t)\ dt/|I|$$

where B is a point–interval *before*-relation. The idea of this definition is to average the point–interval *before*-relation over the interval I. The normalisation factor is just $|I|$. The rationale behind this is the following: if J is finite at the left side then $B(J)(t) = 1$ if t is small enough. If we move the interval I into the area where $B(J)(t) = 1$ for all t in this area then the integral becomes $\int I(t) \cdot B(J)(t)\ dt = \int I(t) \cdot 1\ dt = |I|$, such that $\mathit{before}(I, J) = 1$. Thus, $|I|$ as normalisation factor yields the right result.

The built-in function `integrateAsymmetric`$(I, B(J))$ in GeTS computes the normalised integral $\int I(t) \cdot B(J)(t)\ dt/|I|$.

The next picture illustrates version 3 for two finite crisp intervals. B is the standard point–interval *before* operator of Example 3.3. For this particular operator B and crisp intervals I and J, `IIR::Before`(I, J) yields the percentage of points in I which are before J (in the usual crisp sense).

Example 4.3 (`IIR::Before` for crisp intervals) *The upper dashed line in the picture indicates the fuzzy value at position t if the end of interval I is at this position. The fuzzy value has dropped to 0 only if the interval I is moved completely into J. A steeper decrease can be enforced if the result of* `IIR::Before`(I, J) *is, for example, exponentiated with an exponent > 1. (GeTS has an* `exp` *function for this purpose.) The lower dashed line in the figure therefore indicates* `IIR::Before`$(I, J)^3$.



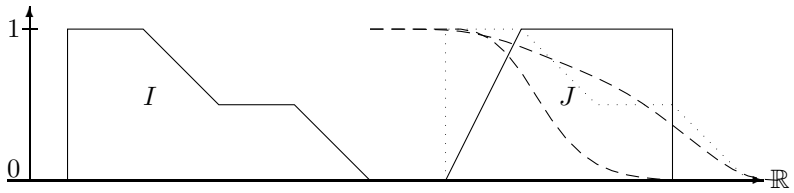
`IIR::Before`(I, J) *and* `IIR::Before`$(I, J)^3$

■

One may argue whether it is desirable to have a 'before' relation where the standard parameters force the fuzzy value down to 0 only when the interval I is completely contained in J. The counter argument is that a smooth decrease can reveal more information about the structure of I and J than a steep drop. A steep drop to 0 can always be achieved by exponentiating the result with a large enough exponent.

The next picture shows the `IIR::Before`–relation for real fuzzy intervals.

Example 4.4 (`IIR::Before` for fuzzy intervals) *The upper dashed line in the picture indicates the result of the* `IIR::Before`*–relation at position t if the positive end of the interval I is moved to t. The lower dashed line is the upper dashed line exponentiated with the exponent 10. The dotted line represents the position of the interval I when the result value is dropped to 0.*



IIR::Before(I, J) *and* IIR::Before(I, J)[10]

■

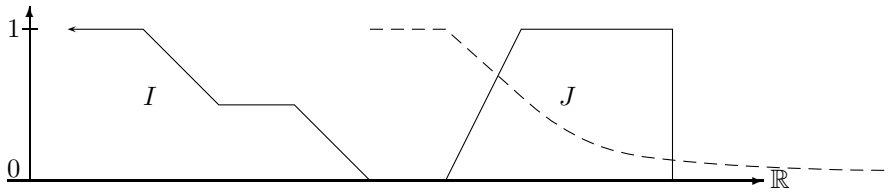
Version 2 for Infinite Intervals
If the interval I is positive infinite, then nothing can be after I. Therefore the relation IIR::Before(I, J) must yield 0. If the interval J is negative infinite, then nothing can be before J. Therefore the relation IIR::Before(I, J) must also yield 0.

If I is negative infinite and J is finite or positive infinite then IIR::Before(I, J) may well be not false. The problem is how to measure the degree of 'beforeness' in this case. Since I is infinite, $\int I(t) \cdot B(J)(t)\ dt$ will always be infinite. An alternative is to take instead of I only the intersection between I and J and to measure the degree of 'beforeness' of $I \cap J$. Since J is not negative infinite, $I \cap J$ is finite. The definition is then

$$\mathit{before}(I, J) \stackrel{\text{def}}{=} \int (I \cap J)(t) \cdot B(J)\ dt / |I \cap J|.$$

This integral is again computed with the GeTS function integrateAsymmetric$(I \cap J, B(J))$

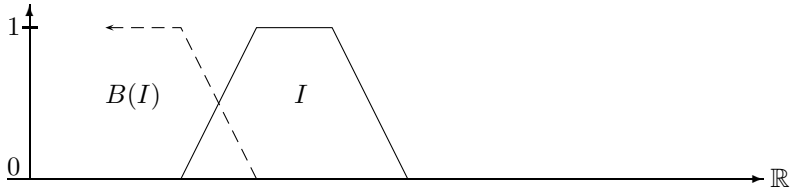
Example 4.5 (IIR::Before for infinite intervals) *This picture shows the development of* IIR::Before(I, J) *when I is negative infinite and its positive end is moved along the time axis. The fuzzy value drops down, but not to 0. It remains constant after a while.*



before *for Infinite Intervals*

■

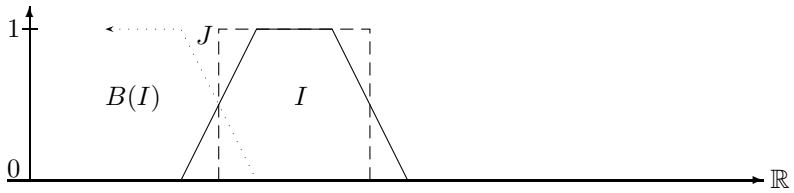
Properties of the IIR::Before–Relation for Finite Intervals:
The crisp relation on crisp sets is irreflexive: $\neg \mathit{before}(I, I)$ holds for all I. IIR::Before$(I, J) = 0$ holds also for crisp sets. IIR::Before$(I, I) = 0$ means in this case that 0% of I is before I itself. This does not longer hold if I is fuzzy. The following picture illustrates the phenomenon:



Counterexample for Irreflexivity

The intersection of $B(I)$ with I is to a certain degree before I. Therefore IIR::Before$(I, I) > 0$.

The crisp *before* relation is asymmetric: $\forall I, J\ \mathit{before}(I, J) \Rightarrow \neg \mathit{before}(J, I)$. A similar property also holds for the IIR::Before–relation on crisp sets: if a non-zero fraction of I is before J then nothing of J can be before I. IIR::Before$(I, J) > 0 \Rightarrow$ IIR::Before$(J, I) = 0$. This does no longer hold for fuzzy sets.



Counterexample for Asymmetry

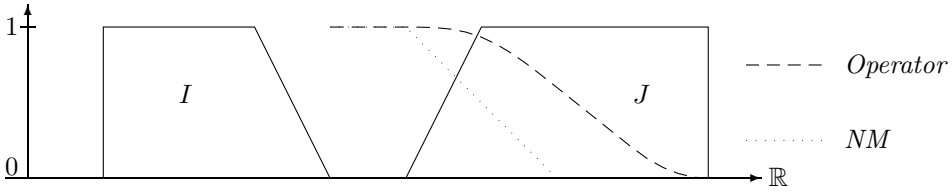
$B(I)$ has a non-zero intersection with J and $B(J)$ has a non-zero intersection with I. Therefore `IIR::Before`$(I, J) > 0$ and `IIR::Before`$(J, I) > 0$.

The fuzzy version of transitivity relates *before*(I, J) and *before*(J, K) with *before*(I, K) in some way. There is such a relation for version 3 of `IIR::Before` and crisp sets: if `IIR::Before`$(I, J) = a$ ($a \cdot 100\%$ of I is before J) and `IIR::Before`$(J, K) = b$ ($b \cdot 100\%$ of J is before K) then one can show that `IIR::Before`$(I, K) = \min(1, a + (b \cdot |I|/|J|))$. Nothing of this kind holds for fuzzy sets.

Version 4 (Nagypál and Motik)

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

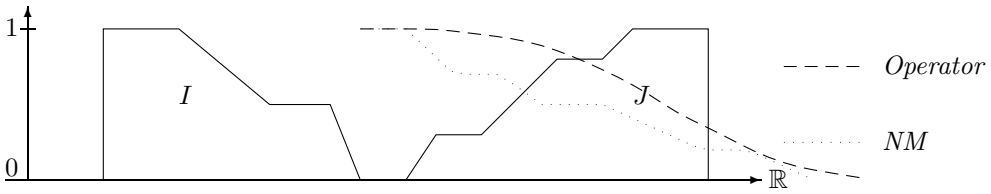
Example 4.6 (`IIR::Before`: operator version versus Nagypál and Motik's version)



`IIR::Before`(I, J) *and* `IIR::NMBefore`(I, J)

It may happen that the result is a non-zero fuzzy value even beyond the positive end of J. This is because even if the positive end of I is behind J, there is still some part of I before J. The next example shows that both versions can show the same phenomenon.

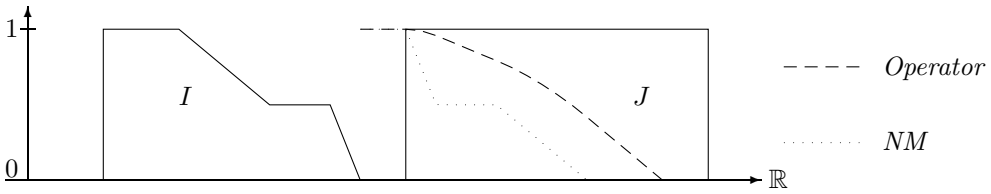
Example 4.7 (`IIR::Before`: operator version versus Nagypál and Motik's version)



`IIR::Before`(I, J) *and* `IIR::NMBefore`(I, J)

The last example is with a fuzzy and a crisp interval. It illustrates that in the Nagypál and Motik version of *before* the structure of the right end of I is mapped directly to the structure of the result.

Example 4.8 (`IIR::Before`: operator version versus Nagypál and Motik's version)



`IIR::Before`(I, J) *and* `IIR::NMBefore`(I, J)

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

4.4 Interval–Interval 'meets' Relations

The classical *meets*–relation yields 'true' if the end of the first interval I touches the beginning of the second interval J. This is essentially version 1 of `IIR::Meets` below. Version 1 has again the region parameter `R` (`support`, `core`, `kernel` or `maximum`) which specifies the crisp region of the intervals to be compared. The other three versions in Def. 4.9 below are the operator definitions and Nagypál and Motik's version.

Definition 4.9 (Interval–Interval 'meets' Relations)

```
1. IIR::Meets(Interval I, Interval J, Region R) =
        case isEmpty(I): false,
             isEmpty(J): false
        else (point(I,right,R) == point(J,left,R))


2. IIR::Meets(Interval I, Interval J, Interval->Interval F,
               Interval->Interval S, Bool simple) =
        case isEmpty(I) or isEmpty(J): 0.0,
             isInfinite(I,right) or isInfinite(J,left): 0.0
        else integrateSymmetric(F(I),S(J),simple)

3. IIR::Meets(Interval I, Interval J, Bool simple) =
        IIR::Meets(I,J,PIR::Finishes[Interval],PIR::Starts[Interval],simple)

4. IIR::NMMeets(Interval I, Interval J) =
        case isEmpty(I) or isEmpty(J): 0.0,
             isInfinite(I,right) or isInfinite(J,left): 0.0
        else Let IN = extend(I,negative) in
             Let JP = extend(J,positive) in
               min(inf(union(IN,JP)),
                   inf(union(complement(IN),complement(JP))))
```

■

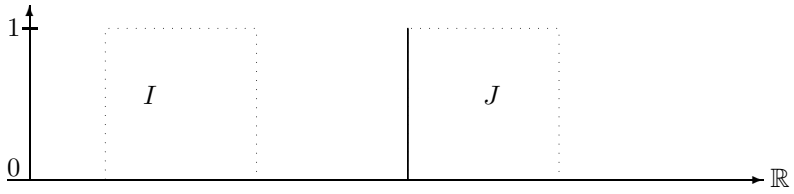
Version 2 and 3

The back end of I and the front end of J are relevant for evaluating $meets(I, J)$. We can get the back end of I in our fuzzy setting with the point–interval `PIR::Finishes` operator (Def. 3.9), and the front end of J with the point–interval `PIR::Starts` operator (Def. 3.6, Example 3.7). The fuzzy *meets*-relation measures how many points in the back end $F(I)$ of I are in the front end $S(J)$ of J and normalises the value with the maximum possible overlap between $F(I)$ and $S(J)$. Notice that this works only if $|F(I)|$ and $|S(J)|$ are finite. The mathematical definition is therefore $meets_{F,S}(I, J) \stackrel{\text{def}}{=} \int F(I)(t) \cdot S(J)(t)\ dt / N(F(I), S(J))$.

The normalisation factor $N(F(I), S(J)) \stackrel{\text{def}}{=} \max_a \int F(I)(t-a) \cdot S(J)(t)\ dt$ amounts to a search problem where $F(I)$ is moved along the time axis to find the position for I where the integral becomes maximal. This guarantees that there is a position for I where $meets(I, J) = 1$. Although the implementation of the search procedure is quite efficient, it may be too expensive or it may be unimportant to normalise the *meets*–relation to 1. Therefore version 2 of `IIR::meets` has a parameter `Bool simple` which is passed to the built-in function `integrateSymmetric`. `integrateSymmetric`$(I, J,$ `simple`$)$ computes $\int I(t) \cdot J(t)\ dt / N(I, J)$ where $N(I, J) = min(|I|, |J|)$ if `simple` = `true`, and $N(I, J) \stackrel{\text{def}}{=} \max_a \int I(t-a) \cdot J(t)\ dt$ if `simple` = `false`. The effect of `simple` = `true` is that there may be no position of I relative to J where `IIR::meets`$(I, J, true) = 1$.

Example 4.10 (`IIR::meets` for crisp intervals)
The first picture shows the `IIR::meets`*–relation where for crisp intervals the operators F and S have a*

singular peak. Consequently the `IIR::meets`*–relation has also a singular peak when the interval I meets J in the crisp sense.*

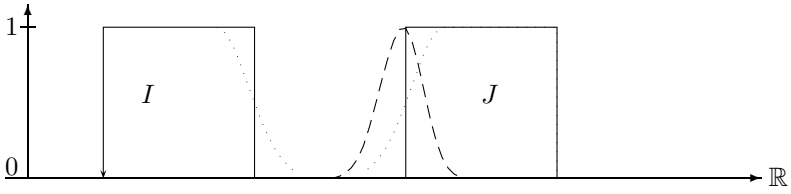


Crisp `IIR::Meets`(I, J) *for Crisp Intervals*

■

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

Example 4.11 (Fuzzy `IIR::Meets` for crisp intervals)

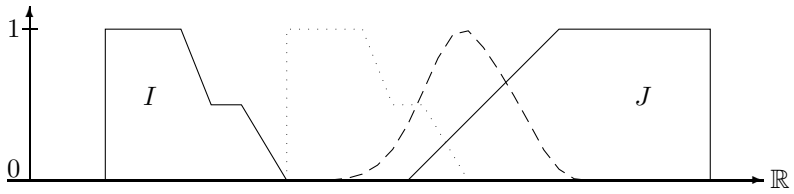


Fuzzy `IIR::Meets` *for Crisp Intervals*

■

Finally we illustrate the fuzzy *meets*-relation with two fuzzy time intervals and the simple point–interval *finishes* and *starts* operators of Example 3.7.

Example 4.12 (Fuzzy `IIR::Meets` for fuzzy intervals) *The dashed line shows the results of the* `IIR::Meets`*–relation when the interval I is moved along the time axis. The dotted figure is the position of I where* `IIR::Meets` *is maximal.*



`IIR::Meets`(I, J) *for Fuzzy Intervals*

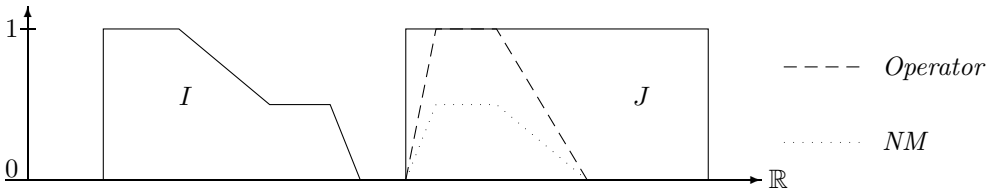
■

Properties of `IIR::Meets`
If the operators F and S have a singular peak then *meets* behaves for crisp relations like the classical crisp *meets*–relation. Therefore the properties of the crisp *meets*–relation hold as well: irreflexivity and asymmetry holds, and transitivity does not hold. If F and S widen the peaks then nothing of this can be predicted any more.

Version 4 (Nagypál and Motik)

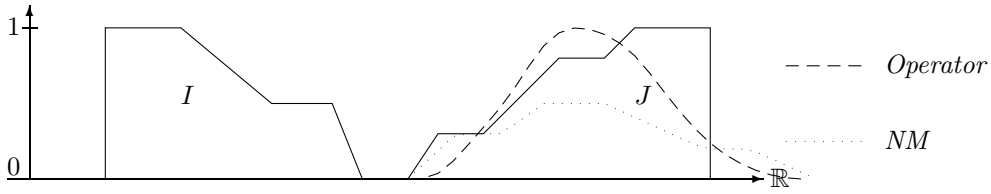
The next examples illustrate that the differences between the operator version of the *meets*–relation and Nagypál and Motik's version are minor. The operator version yields a smoother curve, and the fuzzy values are normalised to 1 as peak value. This would be possible for the Nagypál and Motik version either, but it causes a similar search problem as for the operator version.

Example 4.13 (`IIR::Meets`: operator version and Nagypál and Motik's version)



`IIR::Meets`(I, J) *and* `IIR::NMMeets`(I, J)

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



`IIR::Meets`(I, J) *and* `IIR::NMMeets`(I, J) ■

4.5 Interval–Interval 'overlaps' Relations

The classical relation I *overlaps* J holds if two conditions are fulfilled:
1. a non-empty part I' of I must lie before J, and
2. the rest $(I \setminus I') \neq \emptyset$ must lie inside J.

The generalisation to non-convex intervals concerns the first condition. There are the two possibilities:
1. I' consists of one or more components which lie before J, i.e. there is a gap between the end of I' and the start of J;
2. the last component of I' overlaps with J. This condition is realized in the current implementation of Version 1 of `IIR::Overlaps` below. The 'overlaps' test for non-convex intervals is quite involved. Therefore it is realized as a built-in function `doesOverlap` in GeTS.

Definition 4.14 (Interval–Interval 'overlaps' Relations)

```
1. IIR::Overlaps(Interval I, Interval J, Region R) = doesOverlap(I,J,R)

2. IIR::Overlaps(Interval I, Interval J, Interval->Interval E,
                 (Interval*Interval)->Float D) =
        case isEmpty(I) or isEmpty(J) or isInfinite(J,left) : 0.0,
             isInfinite(I,right) and isInfinite(J,right):
                float(doesOverlap(I,J,support))
             isInfinite(I,right): 0.0,
             isInfinite(J,right): float(doesOverlap(I,J,support))
        else Let EJ = E(J) in
                (1.0 - D(I,EJ)) * D(I,J) /  NormalizeOverlaps(I,J,EJ,D)

3. IIR::Overlaps(Interval I, Interval J)
    = IIR::Overlaps(I,J,lambda(Interval K) extend(K,positive),
                    IIR::During[Interval*Interval])

4. IIR::NMOverlaps(Interval I, Interval J) =
        if(isEmpty(I) or isEmpty(J)) then 0.0
        else Let IL = extend(I,negative) in
             Let IR = extend(I,positive) in
             Let JL = extend(J,negative) in
             Let JR = extend(J,positive) in
                min(sup(intersection(IR,complement(JR))),
                    sup(intersection(IL,JR)),
                    sup(intersection(complement(IL),JL)))
```

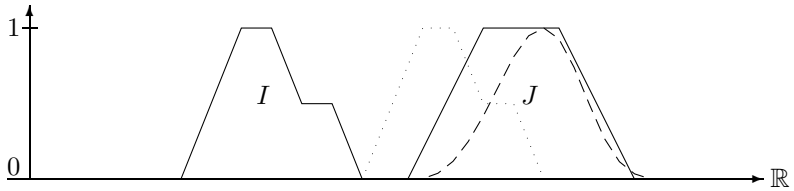
■

Versions 2 and 3

Version 2 needs two operators as parameters for `IIR::Overlaps`. The parameter `E` should be a function that extracts the front part of the interval J and extends it to infinity. `extend(J,positive)` is the natural choice. The operator `D` should be one of the *during*–operators.

Versions 2 of `IIR::Overlaps` for Finite Intervals
The first condition, a non-empty part I_1 of I must lie before J, is encoded in the factor $1 - D(I, E(J))$ $D(I, E(J))$ measures the part of I which is after the front part of J. $1 - D(I, E(J))$ then measures the part of I which is before the front part of J. This factor is multiplied with $D(I, J)$ which corresponds to the second condition. It measures to which degree I is contained in J. The product is normalised with $\max_a((1 - D(shift(I, a), E(J))) \cdot D(shift(I, a), J))$ The normalisation factor corresponds to the maximal possible overlap when I is shifted along the time axis. This guarantees that there is a position for I where $\texttt{IIR::Overlaps}(I, J) = 1$. The normalisation factor is computed by the built-in function `NormalizeOverlaps`.

Example 4.15 (`IIR::Overlaps` for fuzzy intervals) *This example shows the result of the* `IIR::Overlaps` *relation where the standard* during *operator is used (with the identity function as point–interval* during *operator).*



$\texttt{IIR::Overlaps}(I, J)$

The dashed line represents the result of the overlaps relation for a time coordinate t where the positive end of the interval I is moved to t. The dotted figure indicates the interval I moved to the position where $\texttt{IIR::Overlaps}(I, J)$ *becomes maximal.* ■

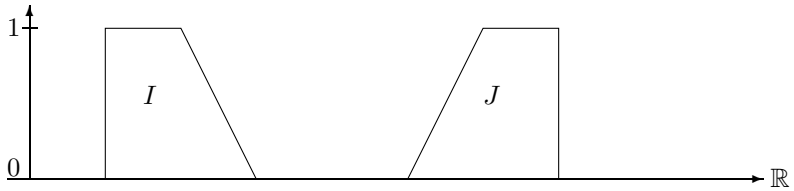
The normalisation factor $\max_a((1 - D(I'(a), E(J))) \cdot D(I'(a), J))$ causes again a search problem. As one can see in the above example the search space is usually very simple (if the intervals are not too exotic). There is only one global maximum and no local maxima. Therefore standard hill climbing is an efficient search method in this case.

Versions 2 of `IIR::Overlaps` for Infinite Intervals
If the interval J is negative infinite then there cannot be anything before J. Therefore $\texttt{IIR::Overlaps}(I, J) = 0$. In the other infinite cases we compute the crisp overlaps relation for the support of I and J.

Properties of the *overlaps* relation:
The crisp *overlaps* relation is irreflexive, asymmetric and not transitive. For version 3 of `IIR::Overlaps` and finite fuzzy sets I which are crisp at the left side we have $D(I, E(I)) = 1$. Therefore,

$$\texttt{IIR::Overlaps}(I, I) = (1 - D(I, E(I)) \cdot D(I, I)/N'(I, J) = 0$$

holds. If I is fuzzy at the left side then $D(I, E(I)) < 1$ and $D(I, I) > 0$, and therefore $\texttt{IIR::Overlaps}(I, I) > 0$.



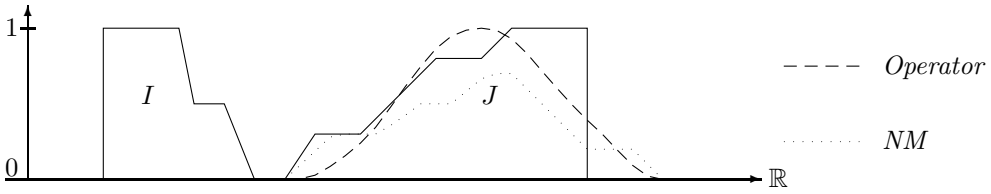
$\texttt{IIR::Overlaps}(I, I) = 0$ *and* $\texttt{IIR::Overlaps}(J, J) > 0$

For crisp intervals I and J we have a property which corresponds to asymmetry: $\texttt{IIR::Overlaps}(I, J) > 0 \Rightarrow \texttt{IIR::Overlaps}(J, I) = 0$. This is because $\texttt{IIR::Overlaps}(I, J) > 0$ means that a part of I is before J. Therefore no part of J can be before I. If I and J are fuzzy, we can have $\texttt{IIR::Overlaps}(I, J) > 0$ and $\texttt{IIR::Overlaps}(J, I) > 0$.

Version 4 (Nagypál and Motik)

The first example below for the *overlaps* relation shows a structural similarity between the operator version and Nagypál and Motik's version. Again, Nagypál and Motik's version is not normalised.

Example 4.16 (`IIR::Overlaps`: operator version versus Nagypál and Motik's version)

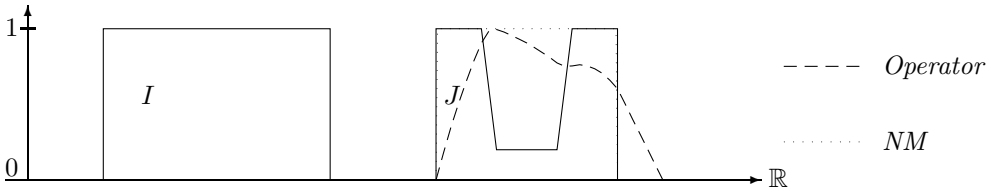


`IIR::Overlaps`(I, J) *and* `IIR::OverlapsNM`(I, J)

The dashed and dotted lines show the result of `IIR::Overlaps` *at a point* t *when the interval* I *is shifted such that its endpoint is at* t*.* ■

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

Example 4.17 (`IIR::Overlaps`: operator version versus Nagypál and Motik's version)



`IIR::Overlaps`(I, J) *and* `IIR::OverlapsNM`(I, J) ■

4.6 Interval–Interval 'starts' Relations

The crisp I *starts* J-relation has two conditions:
1. the start point of I and the start point of J are identical, and
2. I is a subset of J.

These conditions can be generalised straightforwardly to non-convex intervals. They are checked in version 1 of the `IIR::Starts`–relation below.

Definition 4.18 (Interval–Interval 'starts' Relations)

```
1. IIR::Starts(Interval I, Interval J, Region R) =
        if(isEmpty(I) or isEmpty(J)) then false
        else (point(I,left,R) == point(J,left,R)) and isSubset(I,J,R)

2. IIR::Starts(Interval I, Interval J, Interval->Interval S,
                (Interval*Interval)->Float D, Bool simple) =
        case isEmpty(I) or isEmpty(J) or isInfinite(J,left) or
             (isInfinite(I,left) xor isInfinite(J,left)): 0.0,
             isInfinite(I,left) and isInfinite(J,left) : D(I,J)
        else integrateSymmetric(S(I),S(J),simple)*D(I,J)

3. IIR::Starts(Interval I, Interval J, Bool simple) =
      IIR::Starts(I,J,PIR::Starts[Interval], IIR::During[Interval*Interval],simple)

4. IIR::NMStarts(Interval I, Interval J) =
        if(isEmpty(I) or isEmpty(J)) then 0.0
        else Let IR = extend(I,positive) in
             Let JR = extend(J,positive) in
                min(inf(union(complement(IR),JR)),
                    inf(union(IR,complement(JR))),
                    sup(intersection(complement(extend(I,negative)),
                                     extend(J,negative))))
```

■

Versions 2 and 3

The two conditions for the crisp *starts*–relation can be reformulated for the operator version. The conditions are turned into a product of the overlap between the two starting sections of I and J, and the *during*(I, J)-relation:

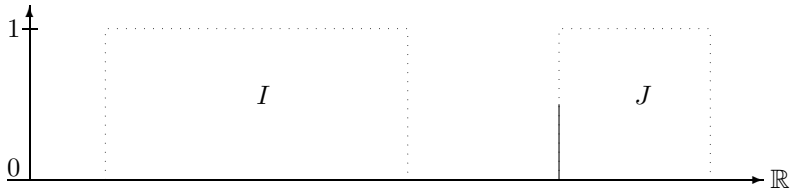
$$starts(I, J) \stackrel{\text{def}}{=} \frac{\int S(I)(t) \cdot S(J)(t)\ dt}{N(S(I), S(J))} \cdot D(I, J)$$

The first factor checks the first condition: the starting part $S(I)$ of I should coincide with the starting part $S(J)$ of J. This value is normalised to the maximal possible overlap of the starting parts. The assumption here is that if I move I along the time axis, there should be a position of I where I definitely starts J, and where therefore the fuzzy value should be 1. The second factor checks whether I is a subset of J. This factor need not be 1 if I is larger than J. Therefore the result of $starts(I, J)$ can be < 1 regardless of the position of I.

The extreme cases are:

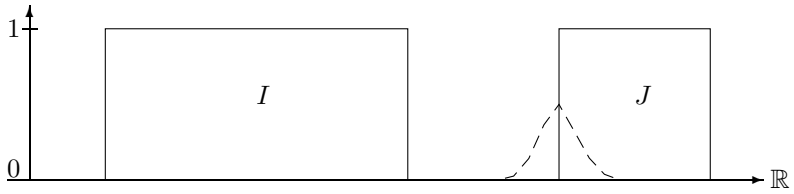
- if I or J are empty then $starts(I, J)$ must be 0;
- if one of I and J is negative infinite then they can't have the same starting point. Therefore $starts(I, J)$ must be 0 again;
- if both are negative infinite then we can assume that they have the same starting point, and therefore only the second condition $during(I, J)$ must be checked;
- it does not matter whether I or J are positive infinite because only the finite starting sections of I and J count.

Example 4.19 (`IIR::Starts` for crisp intervals)
The first picture shows the case where the identity operator is used for D. If the front end of I is moved along the time axis we get a single peak when it meets J. The peak, however, is only 0.5 high because I is twice as large as J.



`IIR::Starts`

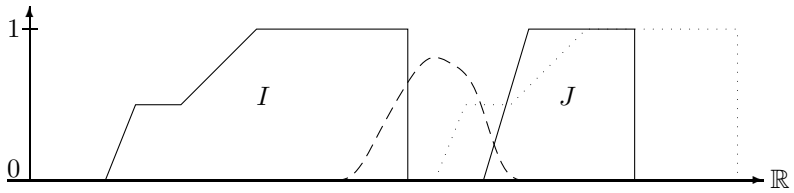
The next picture shows a fuzzified `IIR::Starts`*–relation. The dashed line shows the value of* starts(I, J) *for a position t where the* front end *of I is moved to t. The crisp intervals are fuzzified in the same way as in Example 4.10. The peak is broader, but the maximum is still at 0.5 because I is twice as large as J.*



Fuzzified `IIR::Starts`*–relation*

■

Example 4.20 (`IIR::Starts`–Relation for fuzzy intervals) *The next figure shows the application of the same* `IIR::Starts`*–relation as in the first picture of the above example to fuzzy intervals. The dashed line is again the result of the* `IIR::Starts`*–relation. The dotted figure shows the position of the interval I where* `IIR::Starts`(I, J) *is maximal.*



IIR::Starts*(I,J)*

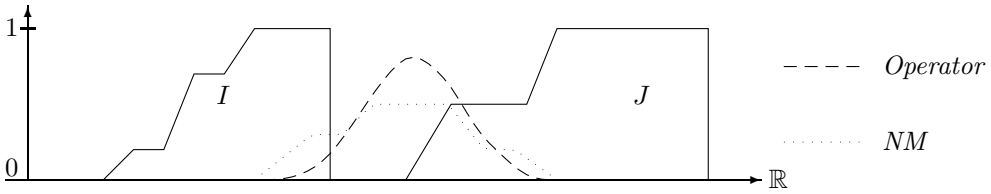
■

Properties of the IIR::Starts(I,J)–relation:
The classical *starts* relation for crisp intervals is reflexive, antisymmetric and transitive. If we consider only the cases $\texttt{IIR::Starts}(I,J) = 0$ or $\texttt{IIR::Starts}(I,J) = 1$ where I and J are crisp intervals then it behaves like the classical starts relation. The same properties hold, in particular reflexivity. For fuzzy intervals, however, we have $0 < \texttt{IIR::Starts}(I,I) < 1$. No kind of fuzzy antisymmetry holds for IIR::Starts on fuzzy intervals. A fuzzy version of transitivity does also not hold. The reason is that, although the starting sections of I and J may overlap, and the starting sections of J and K may overlap, this does not imply that the starting sections of I and K overlap.

Version 4 (Nagypál and Motik)

This version shows close structural similarities if the left sides of the intervals are not too exotic.

Example 4.21 ($\texttt{IIR::Starts}(I,J)$ and $\texttt{IIR::StartsNM}(I,J)$)

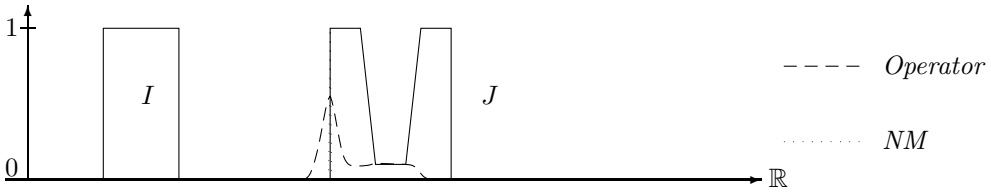


$\texttt{IIR::Starts}(I,J)$ *and* $\texttt{IIR::StartsNM}(I,J)$

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

The Nagypál and Motik *starts*–relation in the next picture has a singular peak of maximal high for the case that the left end of the interval I coincides with the left end of the interval J, although I is not contained in J. The operator version with standard parameters has also a singular peak there, but its hight is only 0.55, which indicates that only 55% of I are contained in J. The dashed line in the picture shows the results of the operator version of *starts*, where the starting sections of I and J are widened by a fuzzification operator.

Example 4.22



$\texttt{IIR::Starts}(I,J)$ *and* $\texttt{IIR::StartsNM}(I,J)$

■

4.7 Interval–Interval 'finishes' Relations

The *finishes*–relation is the mirror image of the *starts*–relation. Therefore we list only the definitions.

Definition 4.23 (Interval–Interval 'finishes' Relations)

```
1. IIR::Finishes(Interval I, Interval J, Region R) =
        case isEmpty(I): false,
             isEmpty(J): false
        else (point(I,right,R) == point(J,right,R)) and isSubset(I,J,R)

2. IIR::Finishes(Interval I, Interval J, Interval->Interval S,
                 (Interval*Interval)->Float D, Bool simple) =
        case isEmpty(I) or isEmpty(J) or isInfinite(J,left) or
                        (isInfinite(I,right) xor isInfinite(J,right)): 0.0,
             isInfinite(I,right) and isInfinite(J,right) : D(I,J)
        else integrateSymmetric(S(I),S(J),simple)*D(I,J)

3. IIR::Finishes(Interval I, Interval J, Bool simple) =
      IIR::Finishes(I,J,PIR::Finishes[Interval], IIR::During[Interval*Interval],simple)

4. IIR::NMFinishes(Interval I, Interval J) =
        if (isEmpty(I) or isEmpty(J)) then 0.0
        else Let IL = extend(I,negative) in
             Let JL = extend(J,negative) in
                min(inf(union(IL,complement(JL))),
                    inf(union(complement(IL),JL)),
                    sup(intersection(extend(I,positive),
                                     complement(extend(J,positive)))))
```

■

4.8 Interval–Interval 'during' Relations

The crisp version of the *during*–relation needs only to check whether an interval I is a subset of an interval J. If I and J are not convex, this is a non-trivial task, but can still be done in linear time with a sweep line algorithm. Therefore it is realized by the built-in function `isSubset` in version 1 of `IIR::During` below.

Definition 4.24 (Interval–Interval 'during' Relations)

```
1. IIR::During(Interval I, Interval J, Region R) = isSubset(I,J,R)

2. IIR::During(Interval I, Interval J, Interval->Interval D) =
        case isEmpty(I): 1.0,
             isEmpty(J): 0.0,
             isInfinite(I):
                Let iNeg=member(point(I,left,support),I) in
                Let iPos=member(point(I,right,support),I) in
                    (iNeg * member(point(J,left,support),I) +
                     iPos * member(point(J,right,support),I)) /
                        (iNeg + iPos)
        else integrateAsymmetric(I,D(J))

3. IIR::During(Interval I, Interval J) =
        IIR::During(Interval I, Interval J, lambda(Interval K) K);

4. IIR::NMDuring(Interval I, Interval J) =
        case isEmpty(I): 1.0,
             isEmpty(J): 0.0
        else min(sup(intersection(complement(extend(I,positive)),extend(J,positive))),
                 sup(intersection(complement(extend(I,negative)),extend(J,negative))))
```

■

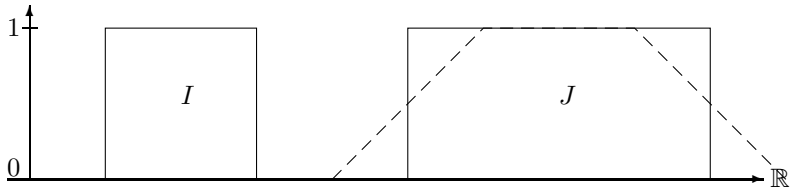
Version 2 and 3

`IIR::During` for Finite Intervals
The operator version of *during* averages the point–interval *during* relation D by integrating over the interval I. The basic formula is therefore

$$during_D(I,J) \stackrel{\text{def}}{=} \frac{\int I(t) \cdot D(J)(t)\ dt}{|I|}$$

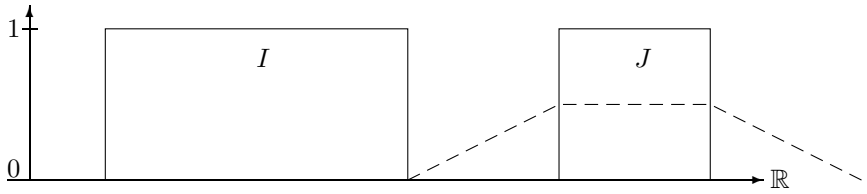
This is realized with a parameter `D` in version 2 of `IIR::During`, and with the identity function for `D` in version 3. `IIR::During`(I,J) measures to what degree I is contained in J. The normalisation factor is $|I|$ because if I is larger than J then `IIR::During`(I,J) should definitely be smaller than 1.

Example 4.25 (`IIR::During` for crisp intervals)
The dashed line in the picture below shows the result of the `IIR::During`*-relation for a coordinate t when the* middle point *of the interval I is moved to t.*



`IIR::During`(I,J) *for Crisp Intervals*

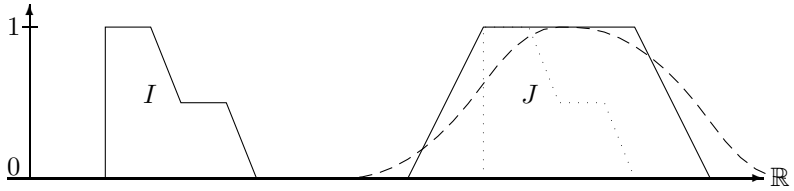
The interval I in the next picture is larger than J such that `IIR::During`(I,J) *never rises to 1.*



`IIR::During`(I,J) *for Crisp Intervals*

■

Example 4.26 (`IIR::During` for fuzzy intervals)
The dashed line shows again the result of the `IIR::During`*-relation when the middle point of I is moved along the time axis. The dotted figure indicates the position of I where* `IIR::During`(I,J) *is maximal.*



`IIR::During`(I,J) *for Fuzzy Intervals*

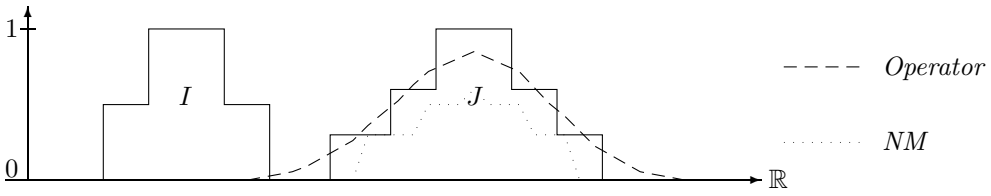
■

`IIR::During` for Infinite Intervals
The formula $\int I(t) \cdot D(J)(t)\ dt/|I|$ cannot be evaluated if I is an infinite interval. Instead one can compute $\lim_{a\to\infty} \int_{-a}^{+a} I(t) \cdot D(J)(t)\ dt / \int_{-a}^{+a} I(t)\ dt$. If the limes is calculated analytically we obtain $(I(-\infty) \cdot J(-\infty) + I(+\infty) + J(+\infty))/(I(-\infty) \cdot I(+\infty))$. This formula is used for version 2 and 3 of `IIR::During` on infinite intervals I.

Version 4 (Nagypál and Motik)

The next two examples show a comparison of the *during*–relations. Compared to the operator version, the Nagypál and Motik version has a much narrower graph and it is less smooth.

Example 4.27 (`IIR::During`: operator version versus Nagypál and Motik's version)

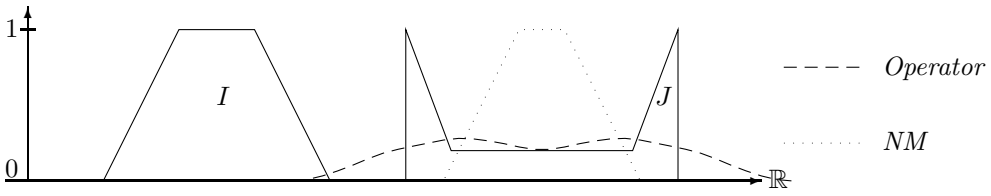


`IIR::During`(I, J) *and* `IIR::NMDuring`(I, J)

The dashed and dotted lines indicate the values of the during relation at coordinate t if the middle point of the interval I is moved to t. ■

The interval J in the next example is treated like its crisp hull by the Nagypál and Motik version. Therefore there is quite a difference to the operator version.

Example 4.28 (`IIR::During`: operator version versus Nagypál and Motik's version)



`IIR::During`(I, J) *and* `IIR::NMDuring`(I, J)

The dashed and dotted lines indicate again the values of the during relation at coordinate t if the middle point of the interval I is moved to t. ■

4.9 Interval–Interval 'equals' Relations

An interval I equals an interval J if I is a subset of J and vice versa. This is what is tested in version 1 below.

Definition 4.29 (Interval–Interval 'equals' Relations)

```
1. IIR::Equals(Interval I, Interval J, Region R) =
       isSubset(I,J,R) and isSubset(J,I,R)

2. IIR::Equals(Interval I, Interval J, (Interval*Interval)->Float D) =
        D(I,J) * D(J,I)

3. IIR::Equals(Interval I, Interval J) = IIR::During(I,J) * IIR::During(J,I)

4. IIR::NMEquals(Interval I, Interval J) =
        case isEmpty(I) : float(isEmpty(J)),
             isEmpty(J) : 0.0
        else Let IL = extend(I,negative) in
             Let IR = extend(I,positive) in
             Let JL = extend(J,negative) in
             Let JR = extend(J,positive) in
                min(inf(union(IL,complement(JL))),
                    inf(union(complement(IL),JL)),
                    inf(union(complement(IR),JR)),
                    inf(union(IR,complement(JR))))
```

■

Version 2 and 3

Version 2 takes an arbitrary binary interval operator D, usually a *during*–operator and computes

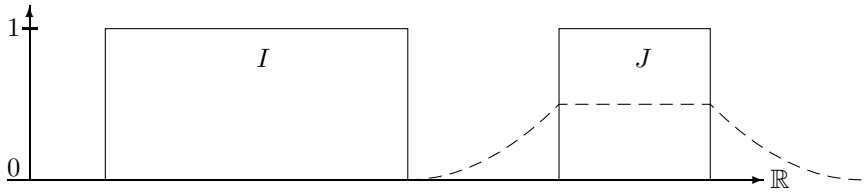
$$equals_D(I, J) \stackrel{\text{def}}{=} D(I, J) \cdot D(J, I).$$

Version 3 uses `IIR::During` for the parameter D.

Example 4.30 (`IIR::Equals` for crisp intervals)
The first picture shows the `IIR::Equals`*–relation for similar intervals. If I is moved on top of J then* $\texttt{IIR::Equals}(I, J) = 1$

`IIR::Equals` *for Similar Intervals*

I and J in the next figure are not equal. Therefore $\text{equals}(I, J)$ *never rises to 1.*



`IIR::Equals` *for Different Intervals*

■

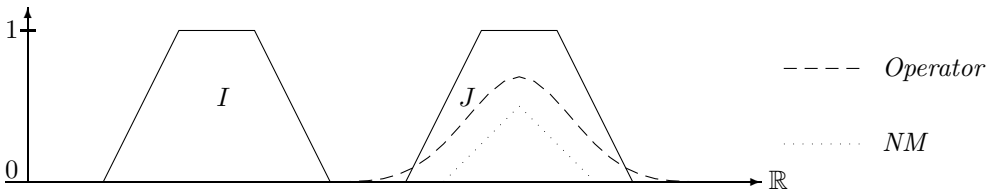
Properties of the `IIR::Equals` relation:
The classical *equals*–relation is an equivalence relation. $\texttt{IIR::Equals}(I, J) = 1$ implies that I and J are crisp and moreover $I = J$ (see the corresponding remark about $\texttt{IIR::During}(I, J) = 1$). Therefore if we consider only the case $\texttt{IIR::Equals}(I, J) = 1$ then `IIR::Equals` is also an equivalence relation.

Since $\texttt{IIR::During}(I, I) > 0$ we also have $\texttt{IIR::Equals}(I, I) > 0$ for non-empty fuzzy sets. By the very definition of `IIR::Equals` we have $\texttt{IIR::Equals}(I, J) = \texttt{IIR::Equals}(J, I)$, the strongest form of fuzzy symmetry. Any form of fuzzy transitivity does not hold. It is easy to construct examples where $\texttt{IIR::Equals}(I, J) > 0$ and $\texttt{IIR::Equals}(J, K) > 0$ and $\texttt{IIR::Equals}(I, K) = 0$. This is because although $\texttt{IIR::Equals}(I, J) > 0$ requires an overlap between I and J, and $\texttt{IIR::Equals}(J, K) > 0$ requires an overlap between J and K, there need not be an overlap between I and K.

Version 4 (Nagypál and Motik)

The behaviour of the *equals* relations are quite similar to the behaviour of the *during* relations because they essentially consist of two *during* relations. The two curves in the first example below do not rise to 1, although the shapes of the two intervals are identical. This is, because the relations do not compare the shapes of the intervals, but they rely on the meaning of the *during* relation.

Example 4.31 (`IIR::Equals`: operator version and Nagypál and Motik's version)

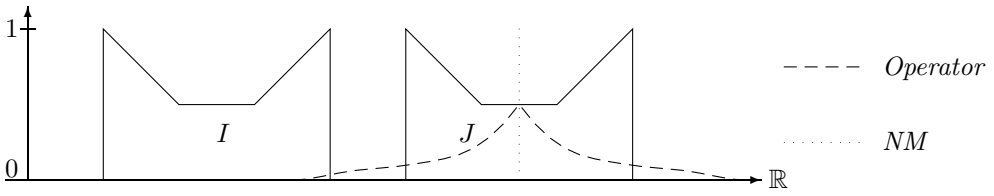


$\texttt{IIR::Equals}(I, J)$ *and* $\texttt{IIR::Equals}(I, J)$

■

The Nagypál and Motik version of the *equals* relation in the next example has a singular peak when the two intervals match exactly. The reason is again that the Nagypál and Motik version treat the intervals in this case like their crisp hulls.

Example 4.32 (`IIR::equals`: operator version and Nagypál and Motik's version)



`IIR::Equals`(I, J) *and* `IIR::Equals`(I, J) ■

4.10 Summary of the Comparison Between the Operator Version and Nagypál and Motik's Relations

The resulting values of the operator version (with standard parameters) and the Nagypál and Motik version of the different relations show a similar structure if the intervals are 'crisp like' intervals, i.e. if they consist of a single component which is more or less monotonic, and has no internal structure. The Nagypál and Motik version follows the rising and falling parts of the intervals more directly, whereas the operator version smoothes the curves. Since the operator versions integrate over the corresponding point-interval relations, they have a more intuitive meaning than the Nagypál and Motik version.

The differences between the two versions are more obvious for crisp intervals, where the Nagypál and Motik versions deliberately behave like the pure crisp relations. The operator versions can return non-trivial fuzzy values is these cases. The differences become also more obvious when the intervals have an internal structure, or when they consist of several components. The internal structure is completely ignored by the Nagypál and Motik versions. This is not the case for the operator version.

Transitivity Table? Allen's interval relations for crisp intervals are related in particular ways. For example, if I *starts* J holds then I *during* J must hold as well. All the relationships between the different interval relations can be collected in a *transitivity table*. The transitivity table can then be used for constraint propagation algorithms. A systematic investigation of the relationships between fuzzy interval–interval relations has not been done yet. The guess is that not many relationships hold, and therefore the transitivity table may not help much for a constraint propagation algorithm.

4.11 Until

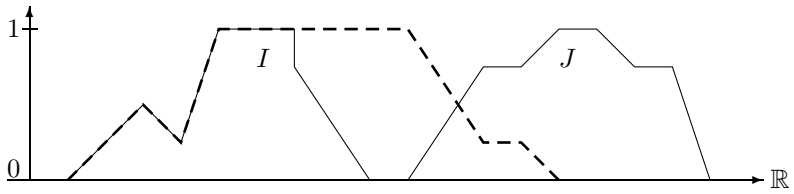
The 'Until' operator is known from temporal logics [3]. φ *Until* ψ in a temporal logic usually means 'eventually ψ holds and φ holds *until* this time point. Since 'Until' is quite useful, we show two GeTS versions of 'Until' where φ and ψ are not formulae, but fuzzy time intervals. With this Until operator we can model expressions like 'from early morning until late night', where 'early morning' and 'late night' are concrete fuzzy intervals. An expression like this is ambiguous. It can be interpreted as 'from the beginning of early morning until the end of late night' or 'from the beginning of early morning until the beginning of late night', and there are two more possibilities. All four combinations are provided. The first version in Def. 4.33 below uses the concrete operators `extend`, `complement` and `intersection`. The second version accepts these operators as extra parameters.

Definition 4.33 (Until)

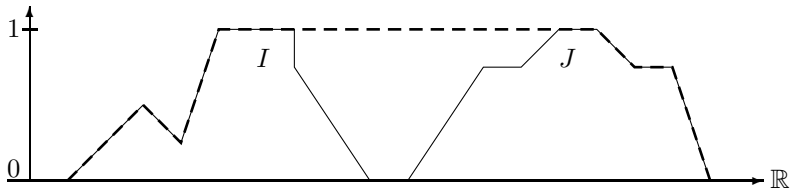
```
1. Until(Interval I, Interval J, Side s1, Side s2) =
   if (s1 == left) then
      (if (s2 == left) then
            intersection(extend(I,positive),complement(extend(J,positive)))
       else intersection(extend(I,positive),extend(J,negative)))
   else
      (if (s2 == left) then
            intersection(complement(extend(I,negative)),complement(extend(J,positive)))
       else intersection(complement(extend(I,negative)),extend(J,negative)));
```

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

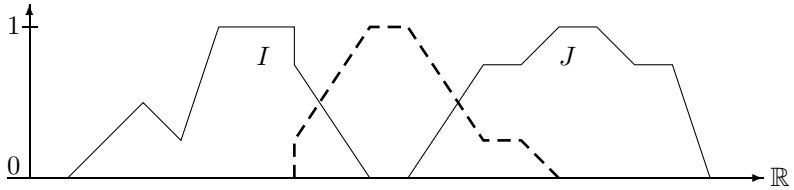
The next four figures show the four cases for the `Until` operator when two single non-overlapping fuzzy intervals are involved.



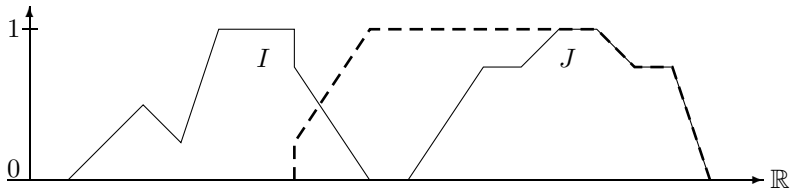
`Until(I,J,left,left)`



`Until(I,J,left,right)`



`Until(I,J,right,left)`

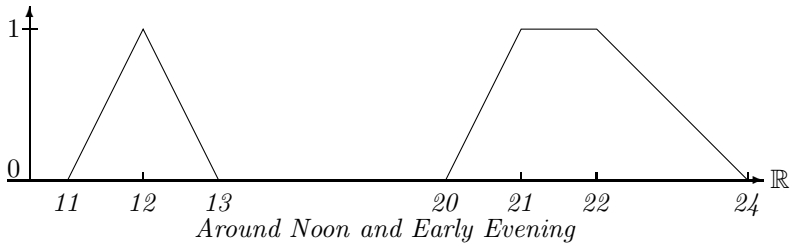


`Until(I,J,right,right)`

The next example shows a concrete application of the second version of `Until` with the following call:

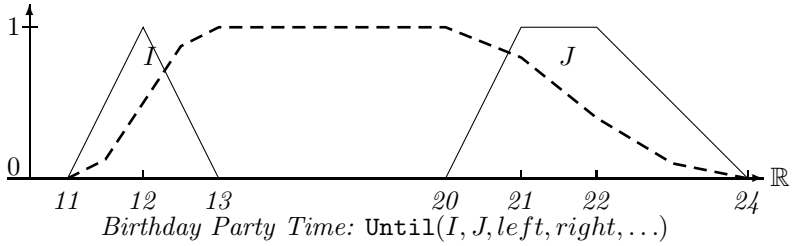
```
Until(I, J, left, right, lambda(Interval K, Interval L) intersection(K,L),
      lambda(Interval K) integrate(K,positive),
      lambda(Interval K) integrate(K,negative),
      lambda(Interval K) complement(K)).
```

Example 4.34 (Birthday Party Time) *The Until operator can be used in more sophisticated ways. Consider a database about, say, the institute's birthday parties. It may contain the entry that the birthday party for the director took place 'from around noon until early evening' of 20/7/2003. 'Around noon' is a fuzzy notion and 'early evening' is a fuzzy notion. Suppose, we have a formalisation of 'around noon' and 'early evening' as the following fuzzy sets:*



Around Noon and Early Evening

What is now the duration of the birthday party? It must obviously also be a fuzzy set. The fuzzy value of the birthday party duration at a time point t is 1 if the probability that the party started before t is 1 and the probability that the party ended after t is also 1. Therefore the fuzzy value at point t is computed by integrating over the probabilities of the start points and the end points. The resulting fuzzy set is:



Birthday Party Time: $\texttt{Until}(I, J, left, right, \ldots)$

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

5 Summary

Different versions of point–interval and interval–interval relations for fuzzy intervals have been presented in this paper. They are defined in the GeoTemporal Specification language (GeTS). The point–interval relations are actually defined as functions which map intervals to intervals. For example, the point–interval *before* function maps an interval I to the interval of all points before I. Three different versions of interval–interval relations are defined. The first version works like the crisp relations by taking crisp regions of the intervals. The second version, the 'operator version' uses suitable point–interval relations and extends the point to an interval by averaging (integrating) over the second interval. The third version is taken from the literature [4]. The effect of the different versions is illustrated graphically with concrete examples. The properties of the operator versions are investigated as far as it is possible. Since the operator version is extremely flexible and can be instantiated in many different ways, an exhaustive investigation of its properties is almost impossible. Therefore, only a few important cases are considered. The operator version is compared with Nagypál and Motik's version by applying them to typical examples. A more general theoretical comparison is not really feasible. The definitions of the point–interval and interval–interval relations are available and can be loaded into the CTTN system.

Acknowledgements

This research has been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. http://rewerse.net).

References

[1] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[2] Didier Dubois and Henri Prade, editors. *Fundamentals of Fuzzy Sets*. Kluwer Academic Publisher, 2000.

[3] D. M. Gabbay, I. Hodkinson, and M Reynolds, editors. *Temporal Logic: Mathematical Foundations and Computational Aspects*, volume 1. Oxford: Clarendon Press, 1994.

[4] 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.

[5] Hans Jürgen Ohlbach. Computational treatment of temporal notions – the CTTN-system. Research Report PMS-FB-2005-30, Inst. für Informatik, LFE PMS, University of Munich, June 2005. URL: http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-30.

[6] Hans Jürgen Ohlbach. Fuzzy time intervals – the FuTI-library. Research Report PMS-FB-2005-26, Inst. für Informatik, LFE PMS, University of Munich, June 2005. URL: http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-26.

[7] Hans Jürgen Ohlbach. GeTS – a specification language for geo-temporal notions. Research Report PMS-FB-2005-29, Inst. für Informatik, LFE PMS, University of Munich, June 2005. URL: http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-29.

[8] Hans Jürgen Ohlbach. Modelling periodic temporal notions by labelled partitionings of the real numbers – the PartLib library. Research Report PMS-FB-2005-28, Inst. für Informatik, LFE PMS, University of Munich, June 2005. URL: http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-28.

[9] L. A. Zadeh. Fuzzy sets. *Information & Control*, 8:338–353, 1965.