A1-D10-1

Implementation: GeTS – A Specification Language for Geo-Temporal Notions

Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Project number:	IST-2004-506779
Project instrument:	EU FP6 Network of Excellence (NoE)
Project thematic priority:	Priority 2: Information Society Technologies (IST)
Document type:	D (deliverable)
Nature of document:	R (report)
Dissemination level:	PU (public)
Document number:	IST506779/Munich/A1-D10-1/D/PU/a1
Responsible editors:	H.J. Ohlbach
Reviewers:	Claude Kirchner
Contributing participants:	Munich
Contributing workpackages:	A1
Contractual date of deliverable:	31 March 2008
Actual submission date:	31 March 2005

Abstract
This document describes the ‘Geo-Temporal’ specification language GeTS. The objects which can be described and manipulated with this language are time points, crisp and fuzzy time intervals and labelled partitionings of the time axis. The partitionings are used to represent periodic temporal notions like months, semesters etc. GeTS is essentially a typed functional language with a few imperative constructs. GeTS can be used to specify and compute with many different kinds of temporal notions, from simple arithmetic operations on time points up to complex fuzzy relations between fuzzy time intervals. The syntax of GeTS together with an operational semantics is described. A parser, a compiler and an abstract machine for GeTS is implemented. The application programming interface for GeTS is documented in the appendix.

Keyword List
temporal notions, specification language

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

Implementation: GeTS – A Specification Language for Geo-Temporal Notions

Hans Jürgen Ohlbach

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

31 March 2005

Abstract
This document describes the 'Geo-Temporal' specification language GeTS. The objects which can be described and manipulated with this language are time points, crisp and fuzzy time intervals and labelled partitionings of the time axis. The partitionings are used to represent periodic temporal notions like months, semesters etc. GeTS is essentially a typed functional language with a few imperative constructs. GeTS can be used to specify and compute with many different kinds of temporal notions, from simple arithmetic operations on time points up to complex fuzzy relations between fuzzy time intervals. The syntax of GeTS together with an operational semantics is described. A parser, a compiler and an abstract machine for GeTS is implemented. The application programming interface for GeTS is documented in the appendix.

Keyword List
temporal notions, specification language

Contents

1 Motivation and Introduction

The phenomenon of *time* has many different facets which are investigated by different communities. Physicists investigate the flow of time and its relation to physical objects and events. Temporal logicians develop abstract models of time where only the aspects of time are formalized which are sufficient to model the behaviour of computer programs and similar processes. Linguists develop models of time which can be used as semantics of temporal expressions in natural language. More and more information about facts and events in the real world is stored in computers, and many of them are annotated with temporal information. Therefore it became necessary to develop computer models of the use of time on our planet, which are sophisticated enough to allow the kind of computation and reasoning that humans can do. Examples are 'calendrical calculations' [4], i.e. formal encodings of calendar systems for mapping dates between different calendar systems. Other models of time have been developed in the temporal database community [2], mainly for dealing with temporal information in databases. This work is becoming more important now with the emergence of the Semantic Web [1]. Informal, semi-formal and formal temporal notions occur frequently in semistructured documents, and need to be 'understood' by query and transformation mechanisms.

The formalisms developed so far approximate the real use of time on our planet to a certain extent, but still ignore important aspects. In the CTTN[1] project [3] we aim at a very detailed modelling of the temporal notions which can occur in semi-structured data. The CTTN system consists of a kernel and several modules around the kernel. The kernel itself consists of several layers. At the bottom layer there are a number of basic datatypes for elementary temporal notions. These are time points, crisp and fuzzy time intervals [8, 9] and partitionings for representing periodical temporal notions like years, months, semesters etc. [10, 11]. The partitionings can be specified algorithmically or algebraically. The algorithmic specifications allows one to encode phenomena like leap seconds, daylight savings time regulations, the Easter date, which depends on the moon cycle etc.

Partitionings can be arranged to form 'durations', e.g. '2 year + 1 month', but also '2 semester + 1 month', where *semester* is a user defined partitioning.

Sets of partitionings, together with certain procedures, form a *calendar*. The Gregorian calendar in particular can be formalized with the partitionings for years, months, weeks, days, hours, minutes and seconds.

A part of the second layer is presented in this paper. It uses the functions and relations of the first layer as building blocks in the specification language GeTS ('GeoTemporal Specifications') for specifying complex temporal notions. A very first version of this language has been presented in [6, 7], but the new version is much more elaborated. It is essentially a functional programming language with certain additional constructs for this application area. A flex/bison type parser and an abstract machine for GeTS has been implemented as part of the CTTN program. GeTS is the first specification and programming language with such a rich variety of built-in data structures and functions for geotemporal notions.

Before the GeTS language is introduced in detail in Sections 3.1 and 3.2, we need to give a short overview over the underlying data structures.

[1]CTTN stands for 'Computational Treatment of Temporal Notions'. The initial working name was actually 'WebCal'. Unfortunately this name had to be given up because of name conflicts with other systems.

2 Basic Data Structures in CTTN

2.1 Time Points and Time Intervals

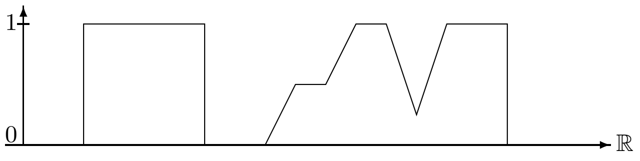
The flow of time underlying most calendar systems corresponds to a time axis which is isomorphic to the real numbers $\mathbb{R}$. Therefore we take as time points just real numbers. Since the most precise clocks developed so far, atomic clocks, measure the time in discrete units, it is sufficient to restrict the representation of concrete time points to *integers*. In the standard setting these integers count the *seconds* from the Unix epoch, which is January 1st 1970. Nothing significant changes in GeTS, however, if the meaning of these integers is changed to count, for example, femtoseconds from the year 1.

The next important datatype is that of time intervals. Time intervals can be crisp or fuzzy. With fuzzy intervals one can encode notions like 'around noon' or 'late night' etc. This is more general and more flexible than crisp intervals. Therefore the CTTN system uses fuzzy intervals as basic interval datatype.

Fuzzy Intervals are usually defined through their membership functions [12, 5]. A membership function maps a base set to real numbers between 0 and 1. The base set for fuzzy time intervals is a linear time axis, isomorphic to the real numbers.

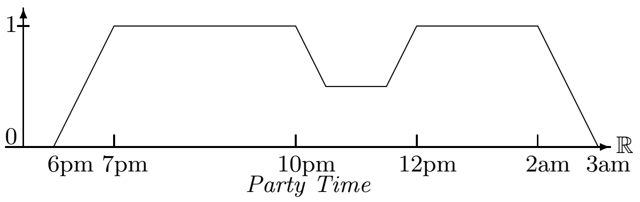
Definition 2.1 (Fuzzy Time Intervals) *A* fuzzy membership function *in GeTS is a total function $f : \mathbb{R} \mapsto [0, 1]$ which does not need to be continuous, but it must be integratable. The* fuzzy interval *I_f that corresponds to a fuzzy membership function f is $I_f \overset{\text{def}}{=} \{(x, y) \subseteq \mathbb{R}\times[0, 1] \mid y \leq f(x)\}$. Given a fuzzy interval I we usually write $I(x)$ to indicate the corresponding membership function.* ■

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



Crisp and Fuzzy Intervals

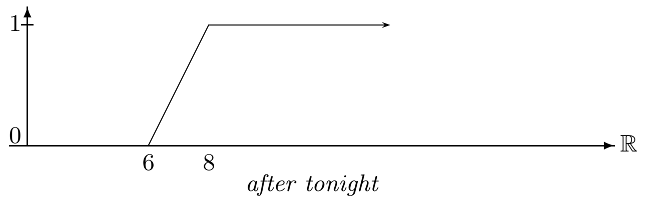
It also comprises finite fuzzy intervals like this one:



Party Time

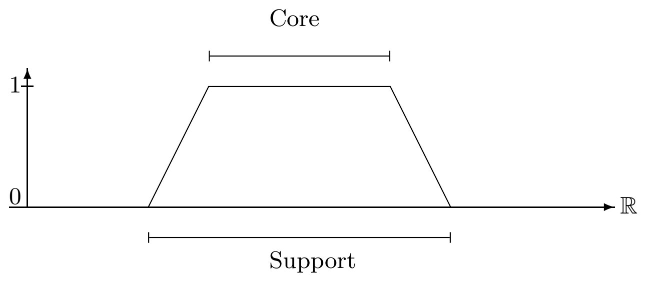
This set may represent a particular party time, where the first guests arrive at 6 pm. At 7 pm all guests are there. Half of them disappear between 10 and 12 pm (because they go to the pub next door to watch an important soccer game). Between 12 pm and 2 am all of them are back. At 2 am the first ones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

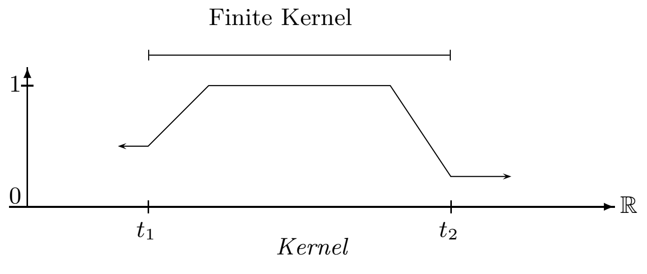


after tonight

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, i.e. the kernel is the smallest convex interval in $\mathbb{R}$ such that $I(x) = y_1$ for all x before the kernel and $I(x) = y_2$ for all x after the kernel. 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.



Core and Support



Kernel

Notice that neither the support nor the core of a fuzzy interval need be convex intervals. The kernel, however, is always a convex interval.

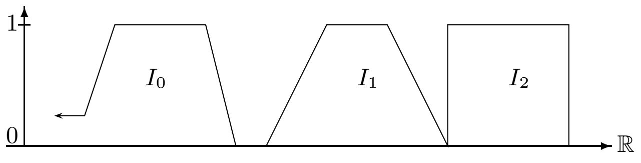
Components
Fuzzy time intervals can consist of several different components. A component is a sub-interval of a fuzzy interval such that the left and right end is either the infinity, or the membership function drops down to 0.

Definition 2.2 (Components) *Let I be a fuzzy time interval. The* components $I_0, \ldots, I_n$ *of*

I are the largest set of subsets of I such that: (i) for $0 \leq k \leq n$: $I_k(x) = I(x)$ for all x in the support of I_k, and (ii) for all $k \in \{1, \ldots, n-1\}$: $I_k^{lS} = -\infty$ or $(\lim_{x \to I_k^{lS}} I(x) = 0$ or $\lim_{I_k^{lS} \leftarrow x} I(x) = 0)$ and $I_k^{rS} = +\infty$ or $(\lim_{x \to I_k^{rS}} I(x) = 0$ or $\lim_{I_k^{rS} \leftarrow x} I(x) = 0)$, where I_k^{lS} is the left boundary of I_k's support, and I_k^{rS} is the right boundary of I_k's support. ■

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

Example:



Components

More features of fuzzy time intervals are introduced when the corresponding language constructs of GeTS are introduced.

2.2 Partitionings

The CTTN system uses the concept of *partitionings* of the real numbers to model periodical temporal notions. In particular, the basic time units years, months etc. are realized as partitionings. Other periodical temporal notions, for example semesters, school holidays, sunsets and sunrises etc. can also be modelled as partitionings.

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

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

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

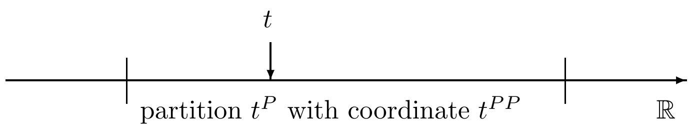
Definition 2.3 (Partitionings) *A partitioning P of the real numbers $\mathbb{R}$ is in CTTN an infinite sequence*
$\ldots [t_{-1}, t_0[, [t_0, t_1[, [t_1, t_2[, \ldots$
of half-open non-empty intervals in $\mathbb{R}$ with integer boundaries. Each interval $[t_i, t_{i+1}[$ is a partition *of the time line.*

For a time point t and a partitioning P let t^P be the P-partition containing t.

A coordinate mapping *of a partitioning P is a bijective mapping between the intervals in P and the integers. Since we always use one single coordinate mapping for a partitioning P, we can just use P itself to indicate the mapping. Therefore let p^P be the* coordinate *of the partition p in P.*

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

For a time t let $t^{PP} \stackrel{\text{def}}{=} (t^P)^P$ be the coordinate of the P-partition containing t. ■



Remark 2.4 (Calendar Systems) *A* calendar *in the CTTN system is a set of partitionings, for example the partitionings for seconds, minutes, hours, weeks, months and years, together with some extra data and methods. Calendars are not visible in the GeTS language because they are only special cases of sets of partitionings. Some GeTS constructs use partitionings which can not only be the partitionings of calendar systems, but any kind of partitioning. This is more general than sticking to particular calendar systems.* ■

Remark 2.5 (Finite Partitionings) The partitionings in CTTN can represent infinite partitionings of the real numbers. This is suitable to model, for example, years. They can, however, also be used to represent *finite* sequences of intervals. Examples are the school holidays in Bavaria from 1970 until 2006. CTTN extrapolates these intervals in a certain way to get an infinite partitioning. This simplifies the algorithms considerably, but it may yield unwanted results for time points where the partitioning is not meant for.

Therefore one can define boundaries for the validity of the partitionings. These boundaries have no influence on the computations, but they can be checked with special functions in the GeTS language (Def. 3.68). ■

The CTTN system uses *labelled partitionings.* The labels are names for the partitions. They can be used for two purposes. The first purpose is to get access to the partitions via their names (labels). For example, the labels for the 'day' partitioning can be 'Monday', 'Tuesday' etc., and one can use these names in various GeTS functions. The second purpose is to use the labels to group partitions together to so called *granules.* The concept of 'working day, for example, can be modelled by taking an 'hour partitioning, and attach labels 'working_hour' and 'gap' to the hour partitions. Groups of hour partitions labelled 'working_hour' yield a working day. The working days can be interrupted by 'gap partitions, for example to take 'lunch time out of a 'working day. Example 3.27 below illustrates the concept of granules in more detail.

Definition 2.6 (Labels and Granules) *A* labelling *L is a finite sequence of strings $l_0, \ldots, l_{n-1}$. The label* gap *has a special meaning.*

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

A granule *is a sequence $p_i, \ldots, p_{i+k}$ of partitions such that: (1) the labels of p_i and p_{i+k} are not* gap*; (2) the labels of $p_i, \ldots, p_{i+k}$ which are not* gap *are the same, and (3) $i \bmod n < (i+k) \bmod n$.* ■

Example 2.7 (The Labelling of Days) The origin of the reference time is again January 1st 1970. This was a Thursday. Therefore we choose as labelling for the day partitioning

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

The following correspondences are obtained:

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

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

2.3 Durations

The partitionings are the mathematical model of periodic time units, such as years, months etc. This offers the possibility to define *durations.* A duration may, for example, be '3 months + 2 weeks'. Months and weeks are represented as partitionings, and 3 and 2 denote the number of partitions in these partitionings. The numbers need not be integers, but they can be arbitrary real numbers.

A duration can be interpreted as the length of an interval. In this case the numbers should not be negative. A duration, however, can also be interpreted as a time shift. In this interpretation negative numbers make perfect sense. $d = -2\ \mathit{week} + 3\ \mathit{month}$, for example, denotes a backward shift of 2 weeks followed by a forward shift of 3 months.

Definition 2.8 (Duration) *A* duration $d = d_0\ P_0, \ldots, d_k\ P_k$ *is a list of pairs where the d_i are real numbers and the P_i are partitionings.*

If a duration is interpreted as a shift of a time point, it may be necessary to turn the shift around, in the backwards direction. Therefore the inverse of a duration is defined:
$-d \overset{\text{def}}{=} -d_k\ P_k, \ldots, -d_0\ P_0$
We also need a negated duration where the order of the partitionings stays the same:
$neg(d) \overset{\text{def}}{=} -d_0\ P_0, \ldots, -d_k\ P_k.$ ■

For example, if
$d = 3\ \mathit{month}, 2\ \mathit{week}$ then
$-d = -2\ \mathit{week}, -3\ \mathit{month}$ and
$neg(d) = -3\ \mathit{month}, -2\ \mathit{week}$.

2.4 Date Formats

A *date format* in CTTN specifies the structure of date strings.

Definition 2.9 (Date Format) *A* date format *DF is a sequence $P_0/\ldots/P_k$ of partitionings. A* date *in a date format DF is a sequence $d_0/\ldots/d_n$ of integers with $n \leq k$.* ■

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

3 The GeTS Language

The design of the GeTS language was influenced by the following considerations:

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

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

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

4. The last aspect, but even more the point before, namely that GeTS is to be integrated into a host system, were the main arguments against an easy solution where GeTS is only a particular module in a functional language like SML or Haskell. The host system was developed in C++. Linking a C++ host system to an SML or Haskell interpreter for GeTS would be more complicated than developing GeTS in C++ directly. The drawback is that features like sophisticated type inferencing or general purpose data structures like lists or vectors are not available in the current version of GeTS. If it turns out that they are useful for some applications, however, it is not a big deal to integrate them into GeTS.

5. Developing GeTS from scratch instead of using an existing functional language has also an advantage. One can design the syntax of the language in a way which better reflects the semantics of the language constructs. This makes it easier to understand and use. As an example, the syntax for a time interval constructor is just $[expression_1, expression_2]$. The freedom in designing a nice syntax is, however, is limited by the available parser technology (in the GeTS case, flex and bison). Therefore some of the language features are compromises between intuitiveness and technical constraints.

The GeTS language is a strongly typed functional language with a few imperative constructs. Let us get a flavour of the language, before the technical details are introduced.

Example 3.1 (tomorrow) The definition

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

specifies 'tomorrow' as follows: `now()` yields the time point of the current point in time (Def. 3.25). `day` is the name of the day partitioning. Let i be the coordinate of the day-partition containing `now()`. `partition(now(),day,1,1)` computes the interval $[t_1, t_2[$ where t_1 is the start of the partition with coordinate $i + 1$ and t_2 is the end of the partition with coordinate $i + 1$. Thus, $[t_1, t_2[$ is in fact the interval which corresponds to 'tomorrow'.

In a similar way, we can define

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

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

Example 3.2 (Christmas) The definition

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

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

`date(t,Gregorian_month)` computes a date representation for the time point `t` in the date format `Gregorian_month` (`year/month/day/hour/minute/second`). Only the year is needed.

`dLet year = ...` therefore binds only the year to the integer variable `year`. If, for example, in addition the month is needed one can write `dLet year|month = date(...`.

`time(year|12|25,Gregorian_month)` computes t_1 = begin of the 25th of December of this year. `time(year|12|27,Gregorian_month)` computes t_2 = begin of the 27th of December of this year. The expression `[...,...]` denotes the half-open interval $[t_1, t_2[$.[2] The result is therefore the half-open interval from the beginning of the 25th of December of this year until the end of the 26th of December of this year.

Example 3.3 (Point–Interval Before Relation) The function

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

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

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

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

Example 3.4 (Fuzzy Interval–Interval Before Relation) A fuzzy version of an interval–interval before relation could be

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

■

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

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

The next example illustrates some procedural features of GeTS. The `effect` function takes two intervals and a function `F`, which maps the two intervals to a fuzzy value. `F` could for example be a fuzzy interval–interval relation. The first interval `I` is now shifted `step` times by the given `distance`, and each time `F(I,J)` is computed. These values are inserted into a new interval, which is the result of the function.

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

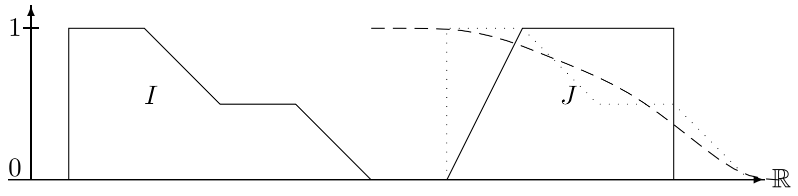
Example 3.5 (effect)

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

■

'`Let K = []`' creates a new empty interval and binds it to the variable `K`. The `while` loop shifts the interval `I` `steps` times by the given `distance` (`I := shift(I,distance)`). Each time `pushBack(K,point(I,right,kernel),F(I,J))` adds the pair (x, y) consisting of x = right boundary of the kernel of the shifted `I` and y = `F(I,J)` to the interval `K`.

The dashed line in the figure below shows the result of the `effect` function when applied to the two intervals `I` and `J`, and a suitable interval–interval 'before' relation as parameter `F`. The dotted figure shows the position of the shifted interval `I` when the `F(I,J)` drops down to 0.



Effect of the `effect` *function*

3.1 Types in the GeTS Language

The GeTS language has a fixed number of basic types. They represent certain data structures and certain keywords. So far there is no mechanism for extending the basic types. The basic types can be combined to functional types $T_1 * \ldots * T_n \mapsto T$.

3.1.1 Basic Types

There are two groups of basic types, the data structure types and the enumeration types. The data structure types represent built in data structures.

Definition 3.6 (Data Structure Types)

`Integer`	*standard integers*
`Time`	*very long integers*
`Float`	*standard floating point numbers*
`String`	*strings*
`Interval`	*fuzzy intervals*
`Partitioning`	*partitionings*
`Label`	*labels for partitions*
`Duration`	*durations*
`DateFormat`	*date formats*

■

The data structure types abstract away from the concrete implementation. The `Integer` type, for example, is currently realized as 32 bit signed integer data, while the `Time` type is currently realized as 64 bit signed integer data. The `Float` type is currently realized by a 32 bit 'float' data type. One should, however, not exploit this in any way.

`String`s are currently just sequences of 8-bit characters. (This may change in future releases to support Unicode).

`Interval`s (Def. 2.1) are realized as *polygons* with integer coordinates. An interval is therefore a sequence of pairs $I = (x_0, y_0), \ldots, (x_n, y_n)$. The x_i are `Time` points and the y_i are fuzzy values. Internally the y_i are realized as short integers between 0 and 1000. From the GeTS point of view, however, the y_i are `Float` numbers between 0 and 1. The interval I is *negative infinite* if $y_0 \neq 0$. I is *positive infinite* if $y_n \neq 0$. The internal representation of `Interval` data, however, is completely invisible to the GeTS user. Details about the internal representation and the algorithms can be found in [8].

`Partitioning`s (Def. 2.3) are complex data structures. Fortunately, this is also not visible to the GeTS user. Partitionings are just parameters to some of the functions. They can be used without knowing anything about the internal details.

`Label`s (Def. 2.6) for partitions are in principle just strings. It is, however, possible to use different strings for the same label. For example, one can label days with English names "Monday", "Tuesday" etc., and with German names "Montag", "Dienstag" etc., and switch between these versions. This is also transparent to the user. Nevertheless, it makes it necessary to consider labels not as strings, but as data structures (see [8] for details).

`Duration`s (Def. 2.8) are sequences of pairs $d_0\ P_0, \ldots, d_n\ P_n$ where the d_i are `Float` data and the P_i are *Partitionings*.

`DateFormat`s (Def. 2.9) are sequences $P_0/\ldots/P_n$ of Partitionings.

The data structure types are used as types for variables, but they can also be used explicitly as constants, so called literals. To this end, there is a string representation of the data structure types. These strings are parsed by the GeTS parser and mapped to the internal representation.

Remark 3.7 (String Representation of Data Structure Types) The data structure types have the following string representation:

`Integer`: sequences of digits, optionally preceded by '+' or '-'. Examples are 123, +4, -345. The length of these sequences depends on the internal representation of integers.

`Time`: sequences of digits, optionally preceded by '+' or '-' and optionally followed by 'T'. Examples: 12345678901, 3T, -23T. The length of these sequences depend on the internal representation of `Time` values.

Notice that a string of digits, which is not followed by 'T' is first parsed as `Integer`. Only if this fails, it is parsed as `Time` data. Therefore the string 123 will always be mapped to `Integer` data, and not to `Time` data. Usually this should not harm, because `Integer` data are automatically casted to `Time` data when this becomes necessary.

`Float`: They have the standard representation of float or double values. Examples are -1.5, 3.4e-2, -77e+5. The length of mantissa and exponent depends on the realization of `Float` values.

`String`: They are arbitrary sequences of characters enclosed in quotes: "characters". The two characters \n are interpreted as newline command. A quote " within the string must be escaped with a \ character. The character sequences "ab\"cd\"ef" is therefore parsed as the string ab"cd"ef.

`Interval`: Intervals cannot be explicitly referenced within a GeTS function definition. The only exception is the empty interval, which is represented by []. The GeTS module, however, provides an interface function which allows one to call GeTS functions with a string representation of the arguments. This function accepts non-negative integers as identifiers for the intervals, together with a vector of pointers to the actual intervals. The integer identifiers are used as indices to this vector.

`Partitioning`: It is assumed that all necessary partitionings are predefined, and can be identified by *names*. The names can be used in GeTS function definitions. The parser maps them to the actual partitioning data structures. The names of the partitionings, which make up the Gregorian calendar, are `year`, `month`, `week`, `day`, `hour`, `minute`, `second`.

`Label`: Labels must also be predefined. They are identified by their name.

`Duration`: The representation of a *simple* duration is $d_0\ P_0 + \ldots + d_n\ P_n$ where the d_i are `Integer` or `Float` expressions and the P_i are partitioning names or variables. Repeating patterns like '2 week + 1 day + 2 week + 1 day' can be abbreviated by '2*(2 week + 1 day)'.

`DateFormat`: They need to be predefined. Date formats are accessed by their names in GeTS specifications. Date formats in CTTN are predefined for each calendar system. The predefined date formats for the Gregorian calendar are `Gregorian_month` and `Gregorian_week`. Thus, GeTS invokes calendar systmes, but only implicitely via the data formats.

■

A number of enumeration types is predefined in GeTS. They are used to control some of the algorithms. Their meaning therefore depends on the meaning of the built-in function where they occur as parameters.

Definition 3.8 (Enumeration Types)

type name	*possible values*
`Bool`	`true/false`
`Side`	`left/right`
`PosNeg`	`positive/negative`
`UpDown`	`up/down`
`IntvRegion`	`core/kernel/support`
`PointRegion`	`core/kernel/support/maximum`
`Hull`	`core/kernel/support/crisp/monotone/convex`
`Fuzzify`	`linear/gaussian`
`Inclusion`	`subset/overlaps/bigger_part_inside`
`SplitInclusion`	`align/subset/overlaps/bigger_part_inside`
`Sequencing`	`sequential/overlapping/with_gaps`
`SDVersion`	`Kleene/Lukasiewicz/Goedel`

■

Notice that, for example, the keyword `core` occurs in the enumeration types `IntvRegion`, `PointRegion` and `Hull`. Which type is meant is determined by the context where it occurs. If the context is not clear, for example in comparisons 'expression == keyword', one can use `Icore` (type `IntvRegion`), `Pcore` (type `PointRegion`), or `Hcore` (type `Hull`). The same holds of the keywords `kernel` and `support`.

An unknown string is parsed in the following way:
1. is it an `Integer` value?
2. is it a `Time` value?
3. is it a `Float` value?
4. is it a keyword of one of the enumeration types?
5. is it a partitioning?
6. is it a date format?

If none of these succeed then a parse error is generated.

Definition 3.9 (Basic Types) *A Basic Type in GeTS is either a* data structure type *(Def. 3.6), an* enumeration type *(Def. 3.8), or the special type* `Void` *for expressions which do not return any values.* ■

Automatic Type Conversion:
Automatic type conversion is done from the type `Integer` to the types `Float` and `Time`. That means, the type `Integer` is also acceptable whenever a type `Float` or a type `Time` is required.

3.1.2 Compound Types

Definition 3.10 (Compound Type) *A compound type in GeTS is an expression $T_1 * \ldots * T_n \mapsto T$ where T and the T_i are either basic types or compound types.*

A type expression *is either a basic type or a compound type expression.* ■

3.2 Language Constructs for GeTS

The GeTS language has a number of general purpose functional and imperative language components. Additionally a number of language constructs are geared to manipulating time points, temporal intervals, partitionings, dates etc. As already mentioned, the language is strongly typed. This means, the type of each expression is determined by the top level function name together with the types of its arguments.

GeTS tries to minimize the required number of parentheses in the expressions. Nevertheless, it is usually clearer and easier to understand when enough parentheses are used.

The language has an operational semantics. It is described more or less formally when the language constructs are introduced. The explanations should be clear enough to understand what the language is able to do.

Some aspects of the language depend on the context where it is used. For example, GeTS itself has no exception mechanisms. Nevertheless, exceptions are thrown and must be catched by the host programming system.

Definition 3.11 (Function Definitions) *A GeTS function definition has one of the forms*

$$\begin{array}{lrcl}
(1) & name & = & expression \\
(2) & name() & = & expression \\
(3) & name(type_1\ var_1, \ldots, type_n\ var_n) & = & expression \\
(4) & type : name(type_1\ var_1, \ldots, type_n\ var_n) & = & expression \\
(5) & type : name(type_1\ var_1, \ldots, type_n\ var_n) & &
\end{array}$$

The five versions of function definitions can have a trailer: '`explanation`*: any string'. The explanation is attached at the newly defined function. It can be accessed by the host system.* ■

Version (1) and (2) are for constant expressions, i.e. the name at the left hand side is essentially an abbreviation for the expression at the right hand side. Version (3) is the standard function definition. The type of the function is $type_1 * \ldots * type_n \mapsto T$ where T is the type of the *expression*. Version (4) declares the range type of the function explicitly. It can be used for recursive function definitions, where the name of the newly defined function occurs already in the body. In this case it is necessary to know the range type of the function, before the *expression* can be fully parsed. The factorial function, for example, must be defined in this way:

```
Integer:factorial(Integer n) = if(n == 0) then 1 else n * factorial(n-1)   (1)
```

Finally, version (5) is a forward declaration. It must be used for mutually recursive functions.

Remark 3.12 (Overloading) *Function definitions can be overloaded. They are distinguished by their argument types, not by the result type. This means, two function definitions*

`f(Integer n) = ...` *and*

`f(Float m) = ...`

yield different functions, whereas the second definition in

`Integer:f(Integer n) = ...` *and*

`Float:f(Integer n) = ...`

overwrites the first one or is rejected. This depends on the global control parameter `GeTS::overwrite`*.* ■

Definition 3.13 (Literals) *Literals are strings which can be interpreted as constants of a certain type. See Remark 3.7 for the string representation of literals.* ■

3.2.1 Arithmetic Expressions

GeTS supports the same kind of arithmetic expressions as many other programming languages. A small difference is the `Time` type, which is integrated in the arithmetics of GeTS.

Definition 3.14 (Binary Arithmetic Expressions)
Let N be a number type (i.e. $N =$ `Integer` or $N =$ `Float` or $N =$ `Time`).
If n and m are valid arithmetic expressions then the following binary operations are allowed:

$n + m$	*(addition)*	$n \% m$	*(modulo)*
$n - m$	*(subtraction)*	$max(n, m)$	*(maximum)*
$n * m$	*(multiplication)*	$min(n, m)$	*(minimum)*
n/m	*(division)*	$pow(n, e)$	*(n^e)*

The types are determined according to the following rules:
for the operators '+', '-', '', '/', max and min:*

`Integer`	$*$	`Integer`	$\mapsto$	`Integer`	`Time`	$*$	`Integer`	$\mapsto$	`Time`
`Float`	$*$	`Integer`	$\mapsto$	`Float`	`Time`	$*$	`Time`	$\mapsto$	`Time`
`Integer`	$*$	`Float`	$\mapsto$	`Float`	`Float`	$*$	`Time`	$\mapsto$	`Time`
`Float`	$*$	`Float`	$\mapsto$	`Float`	`Time`	$*$	`Float`	$\mapsto$	`Time`
`Integer`	$*$	`Time`	$\mapsto$	`Time`					

The last two type patterns mean that the result of operations on mixed `Float` *and* `Time` *values are rounded to* `Time` *values. This makes the operations non-associative:* $1.5 + 1.5 + 1T = 4$, *whereas* $1.5 + 1T + 1.5 = 3$.

`Float` *values are not allowed for the modulo operator %. Therefore the remaining type patterns for % are:*

`Integer`	$*$	`Integer`	$\mapsto$	`Integer`	`Time`	$*$	`Integer`	$\mapsto$	`Time`
`Integer`	$*$	`Time`	$\mapsto$	`Time`	`Time`	$*$	`Time`	$\mapsto$	`Time`

The exponentiation operator $pow(n, e)$ is only allowed for `Integer` *exponents and for* `Float` *or* `Integer` *mantissas.*

`Integer`	$*$	`Integer`	$\mapsto$	`Integer`
`Float`	$*$	`Integer`	$\mapsto$	`Float`.

■

Flat expressions like $a + b + c + d$ without parentheses are allowed. The operator precedence is -, +, /, *, i.e. * binds most. The functions *min* and *max* can also accept more than two arguments.

Definition 3.15 (Unary Arithmetic Expressions) *There are four unary arithmetic operators in GeTS:*

$-n$	$[N \mapsto N]$	*N is any number type*
`float`(b)	[`Bool` $\mapsto$ `Float`]	
`round`(a)	[`Float` $\mapsto$ `Integer`]	
`round`$(a,$`up/down`$)$	[`Float` $*$ `UpDown` $\mapsto$ `Integer`]	

■

$-n$ negates the number n.
n can be an expression of type $N =$ `Integer`, $N =$ `Float` or $N =$ `Time`.

`float`(b) turns a boolean value b into a floating point number:
`float(false)` $= 0.0$ and `float(true)` $= 1.0$.

`round`(a) rounds a `Float` value a to the nearest integer. 1.5 is rounded to 1, 1.51 is rounded to 2. -1.5 is rounded to -1, -1.51 is rounded to -2.
`round`(a, `up`) rounds the `Float` value a up, and
`round`(a, `down`) rounds the `Float` value a down.

Definition 3.16 (Arithmetic Comparisons) *If n and m are arithmetic expressions of type* `Integer`*,* `Float` *or* `Time` *then*

$$(n \quad < \quad m), \qquad (n \quad > \quad m)$$
$$(n \quad <= \quad m), \qquad (n \quad >= \quad m)$$

are the usual arithmetic comparison operators. The result is one of the boolean values `true` *or* `false`*. These operators compare different types, i.e.* $(3.9 <= 4T)$ *yields* `true`*, as expected.* ■

The equality and disequality predicates compare numbers in the expected way, but also every other data type.

Definition 3.17 (Equality and Disequality) *If n is an expression of type T and m is an expression of type Q then*

$$n \quad == \quad m \qquad \textit{and} \qquad n \quad != \quad m$$

are expressions of type `Bool`*.*
$n == m$ yields `true` *iff*

1. *T and Q are one of the number types* `Integer`*,* `Float` *and* `Time`*, and the numbers are equal, i.e.* $4.0 == 4T$ *yields* `true`*. (4T is the long integer in the* `Time` *type.).*
2. *$T = Q$, both are enumeration types, and n and m are the same strings. This means in particular: if $T =$* `Hull`*, $Q =$* `IntvRegion`*, $n =$* `core` *and $m =$* `core` *then $n == m$ yields* `false` *(because $T \neq Q$).*
3. *$T = Q =$* `Interval` *and n and m are the same intervals (i.e. the same polygons).*
4. *$T = Q =$* `Partitioning` *and n and m are pointer-equal partitionings*
5. *$T = Q =$* `Duration` *and n and m are the same durations.*

$n \;!=\; m$ yields `true` *iff $n \;==\; m$ yields* `false`*.* ■

3.2.2 Boolean Expressions

GeTS has the standard Boolean connectives: negation (-), and ('and' or '&&'), or ('or' or '||') and exclusive or ('xor' or '^').

Definition 3.18 (Boolean Expressions) *If a and b are Boolean expressions then*

$$\begin{array}{rclc}
 & & -a & [\texttt{Bool} \mapsto \texttt{Bool}] \\
a & \texttt{and} & b & [\texttt{Bool} * \texttt{Bool} \mapsto \texttt{Bool}] \\
a & \texttt{or} & b & [\texttt{Bool} * \texttt{Bool} \mapsto \texttt{Bool}] \\
a & \texttt{xor} & b & [\texttt{Bool} * \texttt{Bool} \mapsto \texttt{Bool}]
\end{array}$$

are Boolean expressions with the corresponding meaning. ■

Flat Boolean expressions without parentheses are also allowed. The operator precedence is `xor`, `or`, `and`, i.e. `and` binds most.

3.2.3 Control Constructs

GeTS has the obligatory 'if-then-else' construct. In addition there is a `case` construct to avoid the need for nested if-then-elses. A 'while' loop is also available. Since GeTS is a functional language, the `while` construct needs a return value. Therefore in addition to the `while` loop body, it has a separate return expression. In the body, however, only imperative constructs (with return type `Void`) are allowed.

Definition 3.19 (`if-then-else`) *If c is an expression of type* `Bool` *and a and b are expressions of the same type T then*

$$\texttt{if } c \texttt{ then } a \texttt{ else } b$$

is an expression of type T.
Thus, the type of the `if` *construct is in general* $\texttt{Bool} * T * T \mapsto T$.
Exceptions are:

1. *If a is of type* `Float`*, and b of type* `Integer`*, or vice versa, then the integer is casted to* `Float`*. The type of* `if` *is in this case:*
   $\texttt{Bool} * \texttt{Float} * \texttt{Integer} \mapsto \texttt{Float}$ *or* $\texttt{Bool} * \texttt{Integer} * \texttt{Float} \mapsto \texttt{Float}$.

   Example: '`if true then 3 else 4.0`*' yields 3.0 as a* `Float` *number.*

2. *If a is of type* `Time`*, and b of type* `Integer`*, or vice versa, then the integer is casted to* `Time`*. The type of* `if` *is in this case:*
   $\texttt{Bool} * \texttt{Time} * \texttt{Integer} \mapsto \texttt{Time}$ *or* $\texttt{Bool} * \texttt{Integer} * \texttt{Time} \mapsto \texttt{Time}$.

■

Notice that a mix of the type `Time` and the type `Float` is not allowed in the `if` statement.
The definition of the factorial function (1) is a typical example for the use of if-then-else.

Definition 3.20 (`case`) *If $C_1, \ldots, C_n$ are Boolean expressions and $E_1, \ldots E_n$ and D are expressions of the same type T then*

$$\texttt{case } C_1 : E_1, ..., C_n : E_n \texttt{ else } D$$

is an expression of type T. ■

The operational semantics of this `case` construct is: the conditions $C_1, \ldots, C_n$ are evaluated in this sequence. If C_i is the first condition, which yields `true` then E_i is evaluated and its result is returned as the result of `case`. If all C_i evaluate to `false` then the result of D is returned.
Exceptions for the requirement that $E_1, \ldots E_n, D$ are expressions of the same type T are: if $T = \texttt{Float}$ or $T = \texttt{Time}$ then some of the $E_1, \ldots E_n$ and D may have type `Integer`. These integers are automatically casted to `Float` or `Time`.
As in the 'if-then-else' construct, a mix of the types `Time` and `Float` is not allowed in the `case` body.

Definition 3.21 (`while`) *Let C be an expression of type* `Bool`*, $E_1, \ldots, E_n$ expressions of type* `Void` *and 'result' and expression of type T then*

$$\texttt{while } C \ \{E_1, ..., E_n\} \ result$$

is an expression of type T. ■

The operational semantics of this `while` construct is: as long as the evaluation of C yields `true`, evaluate the expressions $E_1, \ldots, E_n$ in this sequence. As soon as C yields `false`, evaluate *result* and return this as value of `while`.
An iterative definition of the factorial function is a typical example where the `while` construct is used.

```
factorial(Integer n) = Let f = 1 in while(n>0){f := f*n, n := n-1} f     (2)
```

This example also illustrates the binding construct `Let` and the assignment operation.

Definition 3.22 (`Let`) *The construct*
`Let` *variable* = *expression*1 `in` *expression*2
of type
T
*evaluates the expression*1*, binds the result to the variable and then evaluates expression*2 *under this binding.*
T *is the type of expression*2. ■

Definition 3.23 (Assignment) *If x is a variable of type T, and E is an expression of type T then $x := E$ is an expression of type* `Void`. ■

This is the usual assignment operation: the result of the evaluation of E is assigned to x.
Exceptions for the requirement that x and E have the same type are: if x has type `Float` or `Time` then E may have type `Integer`. The value is automatically casted to `Float` or `Time`.
Notice that the assignment operation returns no value. It can only occur in the body of the `while` statement.

3.2.4 Functional Arguments

A *function call* in GeTS is an expression $name(argument_1, \ldots, argument_n)$ where '*name*' is either the name of a built-in function, or the name of a previously defined function (or a function with forward declaration), or a variable with suitable functional type.

Since variables can have functional types, and GeTS allows overloading of function definitions, it needs a notation for functional arguments. A functional argument can either be just a variable with appropriate functional type, or a function name with argument type specifications, or a lambda expression. A function name with argument type specifications is necessary to choose among different overloaded functions.

Definition 3.24 (Functional Arguments) *A* functional argument *in GeTS is either*

1. *a variable with the appropriate functional type,*
2. *an expression $name[type_1 * \ldots * type_n]$ of a previously defined function with that name and with argument types $type_1 * \ldots * type_n$, or*
3. *a lambda expression:*
   $lambda(type_1\ variable_1, \ldots, type_n\ variable_n)\ expression.$
   If T is the type of 'expression' then $type_1 * \ldots * type_n \mapsto T$ is the type of the lambda-expression.
   'expression' can contain variables which are lexically bound outside the parameter list of lambda.

■

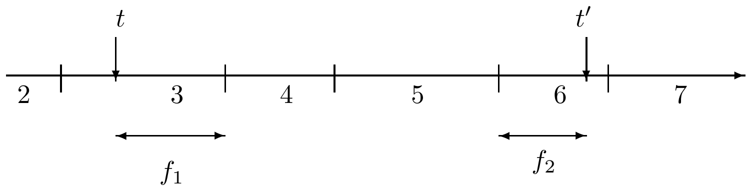
3.2.5 Now and Shift

Definition 3.25 (`now`) *The expression* `now()` *of type* `Time` *yields the current moment in time, i.e. the number of seconds from January, 1st 1970 until the time when the* `now()` *function is invoked.* ■

Notions like ‘in two weeks time’ or ‘three years from now’ etc. denote time shifts. Time shifts are basic operations for many other temporal notions. Therefore GeTS provides a `shift` function which can shift single time points as well as whole intervals by a given duration. Since in general the absolute length of durations depends on the position of the time points at the time axis, shifting time points by *durations* is no trivial operation at all.

A first specification for a `shift` function is to map a time point t to a time point t' such that $t' - t$ is just the required duration, ‘two weeks’ or ‘three years’ in the above examples. This is a *length oriented* `shift` function.

Example 3.26 (for Length Oriented Shift) The algorithm for this function can be best understood by the following example:



Suppose we want to shift the time point t by 3.5 partitions. First, the relative distance f_1 between t and the end of the partition containing t is measured. Suppose it is 0.75. That means from the end of the partition we need to move forward still 2.75 partitions. We can move forward 2 partitions by just adding the 2 to the coordinate 4. We end up at the start of partition 6. From there we need to move forward $f_2 = 0.75$ partitions, which is just 75% of the length of partition 6. ■

Unfortunately the length oriented shift function does not always give intuitive results. Suppose the time point t is noon at March, 15th, and we want to shift t by 1 month. March has 31 days. Therefore the distance to the end of March is exactly 0.5 months. Thus, we need to move exactly 0.5 times the length of April into April. April has 30 days. 0.5 times its length is exactly 14 days. Thus, we end up at midnight April, 14th.

This is not what one would usually expect. We would expect to shift t to the same time of the day as we started with. With the length oriented shift this happens only by chance, or when the partitions have the same length.

GeTS therefore provides also a *date oriented* shift function which avoids the above problems and gives more intuitive results. The idea is to do the calculations not on the level of reference time points, but on the level of dates. If, for example, t represents 2004/2/15, then ‘in one month time’ usually means 2004/3/15. That means the reference time must be turned into a date, the date must be manipulated, and then the manipulated date is turned back into a reference time. This is quite straight forward if the partitioning represents a basic time unit of a calendar system (year, month, week, day etc.), and this calendar system has a date format where the time unit occurs. In the Gregorian calendar this is the case, even for the time unit 'weeks'. 'In two weeks time' requires to turn the reference time into a date format which uses weeks. The corresponding date format uses the counting of weeks in the year (ISO 8601). For example, 2004/42/1 means Tuesday[3] in week 42 in the year 2004. In two weeks time would then be 2004/44/1.

A date oriented shift operation for partitionings which are not standard partitionings of a calendar system can usually be defined by mapping it to a date oriented shift operation for a standard partitioning. For example, if a partitioning 'semester' is defined as sequences of 6 months, one can reduce a shift in terms of semesters to a shift in terms of months. The partitioning module, which underlies the GeTS language, has for each type of partitionings a particular date oriented shift operation. For the details we must therefore refer to [8, 11].

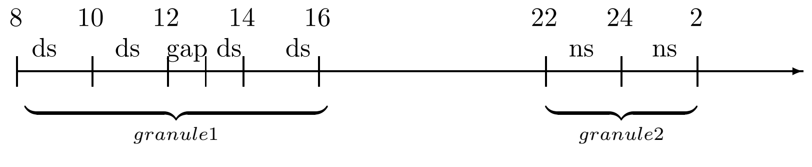
The next problem is to deal with fractional shifts. How can one implement, say, 'in 3.5 months time'? The idea is as follows: suppose the date format is year/month/day/hour/minute/second, and the reference time corresponds to, say, 2004/2/20/10/5/1. First we make a shift by three months and we end up at 2004/5/20/10/5/1. This is a day in May. From the date format we take the information that the next finer grained time unit is 'day'. May has 31 days. $0.5 * 31 = 15.5$. Therefore we need to shift the date first by 15 days, and we end up at 2004/5/34/10/5/1. There is still a remaining shift of half a day. The next finer grained time unit is hour. One day has 24 hours. $0.5 * 24 = 12$. Thus, the last date is shifted by 12 hours, and the final date is now 2004/5/34/22/5/1. This is turned back into a time point.

The date oriented `shift` gives more intuitive results. The drawback is that the distance between the shifted time point and the original time point need no longer be the given duration when it is measured with the `length` function (Def. 3.42).

The `shift` function can not only shift time points by durations like 3.5 months. More complex durations like 3.5 months - 5 days + 3 hours are also admissible for the `shift` function. A shift by such a duration is executed as a sequence of shifts, first by 3.5 months, then by -5 days (backwards shift), and finally by 3 hours.

A statement like 'we must move this task by three working days' refers to a shift of time points which is measured in *granules.* GeTS offers therefore a possibility to shift time points and intervals by durations which are interpreted as *granules.* The basic idea for the algorithm which shifts a time point by a number of granules is to *turn the granules into partitions*, and to use the shift function for partitions. The method is illustrated with the following example:

Example 3.27 Suppose we want to model a *working day* with two shifts, a day shift from 8 am until 4 pm, with a one hour break between 12 am and 1 pm, and a night shift between 10 pm and 2 am. The labelled partitioning is `hour` with labels `ds` (for day shift) and `ns` (for night shift).



The first granule consists of 7 non-gap partitions labelled `ds`. The second granule consists of 4 non-gap partitions labelled `ns`. ■

[3]According to ISO 8601, the first day in a week is Monday. In the standard notation this is day number 1. Since we count days from 0, Monday is day 0 and Tuesday is day 1.

This example shows that shifts by granules yield intuitive results only in special cases. A time point t at 9 am at day n, shifted by 2 granules, should end up at 9 am at day $n+1$. But how can we shift t by one granule, i.e. from the day shift to the night shift? GeTS provides shift operations for this and other cases as well. Whether they yield intuitive results, depends on the application.

A time point t can be: (i) within a non-gap partition of a granule, (ii) within a gap partition of a granule or (iii) within a gap partition between two granules. Suppose we want to shift t by $m = 1.5$ granules. The number of partitions to be shifted is determined as follows:

Case (i): t is within a non-gap partition p of a granule g. Suppose m is positive. Let n be the relative position of p within g (not counting internal gaps). 9 am in Example 3.27 is in partition 1 of a 7-partition granule (the first partition has number 0). $n = 1/7 = 0.1428$. From the start of the first granule we need to move $m + n = 1.6428$ granules forward, i.e. we need to move $n' = 0.6428$ into the second granule. Since this is a 4-partition granule, and $4 \cdot 0.6428 = 2.57$, the target partition is the *third* partition in the second granule. 9 am would be mapped to 1 am next day in the above example.

A negative shift m is treated in a similar way. The only difference is that the relative position of a partition within a granule is computed from the end of the granule and it is represented as a negative number. The relative position of the second partition within a 7-partition granule is in this case $-6/7 = -0.857$.

Case (ii): t is within a gap partition p of a granule g. Let t be at 12:30 am in Example 3.27. If the shift m is integer, we try to move t again into a gap partition of a granule $g + m$. If the granule $g + m$ has gaps, we determined the relative position n of the gap region containing t within g. 12:30 am is in the first gap region of a granule consisting of 2 non-gap regions. Therefore $n = 0.5$. If the target granule $g + m$ has k non-gap regions, the gap region, into which t is to be moved is the $n \cdot k$th gap region. In a second step, the relative position of the gap-partition within the gap region is mapped to a relative position of a gap partition in the target gap-area.

Negative shifts are again treated by computing relative positions as negative numbers, as in case (i).

If the target granule has no gaps, or the shift m is not integer, internal gaps are ignored and the algorithm of case (i) is applied.

Shifting $t =$ 12:30 am by $m = 2$ granules ends up at 12:30 am next day with this method. Shifting $t =$ 12:30 am by $m = 1$ granules ends up at 0:30 am next day.

Case (iii): t is within a gap partition p between two granules g_1 and g_2. If the shift m is integer then the relative position n of the gap partition p within the gap region between g_1 and g_2 is mapped to a relative position of a gap partition p' between the granules $g_1 + m$ and $g_2 + m$. If there are no gaps between $g_1 + m$ and $g_2 + m$ then the target partition p' is just the first partition of the granule $g_2 + m$.

Example 3.27: let t be at 5 pm and $m = 1$. 5 pm is in the second gap-partition of a 6 gap-partition region. The gap-region between the night shift and the day shift consists also of 6 gap-partitions. Therefore t is shifted to the $2/6 \cdot 6 =$ 2nd gap-partition, i.e. 3 am next day.

Positive fractional shifts, for example $m = 1.5$, are treated in a relatively simple way. Let $m' \stackrel{\text{def}}{=} \lfloor m \rfloor$ be the integer part of m. The target partition p' is determined by taking the fractional part $m - m'$ as the relative position of the non-gap partitions of granule $g_2 + m'$. The exact position of t between the two granules plays no role in this case.

Shifting $t = 6:30$ am by 1.5 granules in the above example therefore ends up at 0:30 am next day.

Negative fractional shifts are computed by shifting the end of granule $g_1 + m'$ the fractional part $m - m'$ backwards.

Definition 3.28 (`shift`) *The* `shift` *function can shift a single time point by a given duration:*
`shift`$(time, duration, asGranule, dateOriented)$
is of type
`Time` $*$ `Duration` $*$ `Bool` $*$ `Bool` $\mapsto$ `Time`.

The `shiftLength` *function determines the length of a shift:*
`shiftLength`$(time, duration, asGranule, dateOriented)$
is of type
`Time` $*$ `Duration` $*$ `Bool` $*$ `Bool` $\mapsto$ `Time`.

■

`shiftLength`$(time, duration, asGranule, dateOriented)$ is just an abbreviation for `shift`$(time, duration, asGranule, dateOriented) - time$.

`shift`$(time, duration, asGranule, dateOriented)$ shifts the time point by the given *duration*. If *asGranule* = `true` then the partitionings in the duration are interpreted as granules, otherwise as partitions. If *dateOriented* = `true` then the shift is date oriented, otherwise it is length oriented.

3.2.6 Explicit Construction of Time Intervals

Fuzzy time intervals (type `Interval`) are one of the built-in data structures in GeTS. It is possible to create new empty intervals and fill them up with coordinate points. There are three ways to create new intervals in GeTS:

Definition 3.29 (New Time Intervals)

1. *The expression* [] *stands for the empty interval.*
2. *The expression* $[t_1, t_2]$ *of type* `Time` $*$ `Time` $\mapsto$ `Interval` *constructs a new crisp interval with boundaries* t_1 *and* t_2 *(see Example 3.2).*
3. *The expression* $[(t_1, y_1), (t_2, y_2)]$ *of type* `Time` $*$ `Float` $*$ `TimeFloat` $\mapsto$ `Interval` *constructs a new fuzzy interval with the given two points.*

■

Definition 3.30 (Extending Intervals) *The function*
`pushBack`$(I, time, value)$
of type
`Interval` $*$ `Time` $*$ `Float` $\mapsto$ `Void`
adds the point $(time, value)$ *to the end of the interval* I*.* I *must be an interval which was constructed with* `newInterval()` *(see Def. 3.29). time must lie after the last point in the interval. value must be a* `Float` *value between 0 and 1.* ■

The `pushBack`$(I, time, value)$ function can only ill up the interval I from the past to the future. It throws an error if *time* is before the last time point in I.

3.2.7 Set Operations on Intervals

For crisp intervals there are the standard set operators: complement, intersection, union etc. These are uniquely defined. There is no choice. Unfortunately, or fortunately, because it gives you more flexibility, there are no such uniquely defined set operators for fuzzy intervals. Set operators are essentially transformations of the membership functions, and there are lots of different ones.

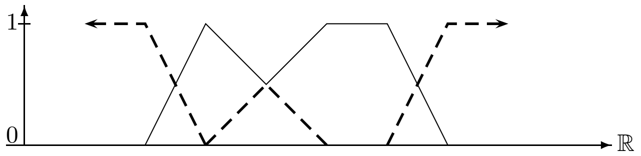
GeTS offers standard versions of the set operators, parameterized set operators of the Hamacher family, and finally set operators with transformation functions for the membership function as parameter. These allow one to customize the set operators in an arbitrary way.

Definition 3.31 (Complement of Intervals)
Let I be an expression of type `Interval`*. The complement operation for intervals comes in three versions:*

(1)	`complement(`I`)`	`Interval` $\mapsto$ `Interval`
(2)	`complement(`I, λ`)`	`Interval` $*$ `Float` $\mapsto$ `Interval`
(3)	`complement(`$I, negation\_function$`)`	`Interval` $*$ (`Float` $\mapsto$ `Float`) $\mapsto$ `Interval`

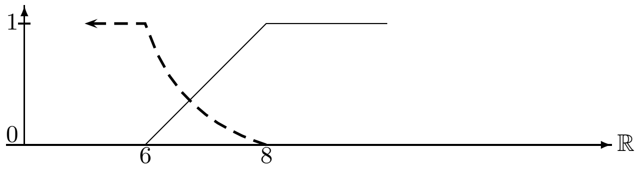
■

Version (1) is the standard complement. Each point (x, y) of the membership function of I is turned into $(x, 1 - y)$.



Standard Complement for a Fuzzy Interval

Version (2) is the *lambda-complement.* For $\lambda > -1$, each point (x, y) of the membership function of I is turned into $(x, \frac{1-y}{1+\lambda y})$. The ordinary complement is computed for $\lambda \leq -1$.



λ-Complement for $\lambda = 2$

Finally, with version (3) it is possible to submit a user defined negation function. For example, with

```
lambda_complement(Interval I, Float lam)
= complement(I,lambda(Float y) (1-y)/(1+lam*y))
```

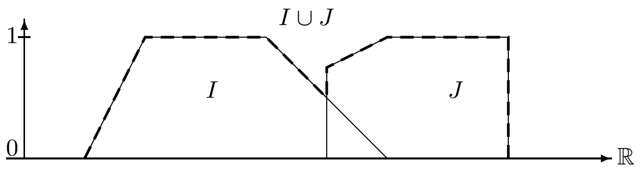
one can define the same lambda-complement with a user defined negation function.

Definition 3.32 (Union of Intervals) *Let I and J be expressions of type* `Interval`. *The union operation for intervals comes in three versions:*

(1) $\texttt{union}(I, J)$ $\qquad \texttt{Interval} * \texttt{Interval} \mapsto \texttt{Interval}$
(2) $\texttt{union}(I, J, \beta)$ $\qquad \texttt{Interval} * \texttt{Interval} * \texttt{Float} \mapsto \texttt{Interval}$
(3) $\texttt{union}(I, J, co\_norm)$
$\texttt{Interval} * \texttt{Interval} * (\texttt{Float} * \texttt{Float} \mapsto \texttt{Float}) \mapsto \texttt{Interval}$

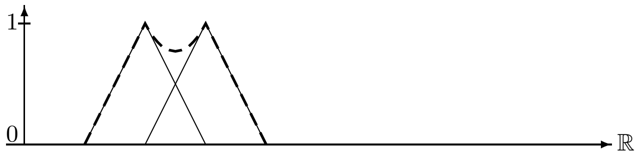
■

Version (1) is the standard union. Each pair (x, y_1) and (x, y_2) of points of the membership function of I and J is turned into $(x, max(y_1 - y_2))$.



Standard Union of Fuzzy Sets

Version (2) is the so called *Hamacher–Union.* For $\beta \geq -1$, each pair (x, y_1) and (x, y_2) of points of the membership function of I and J is turned into $(x, \frac{y_1+y_2+(\beta-1)y_1y_2}{1+\beta y_1 y_2})$. The ordinary union is computed for $\beta < -1$.



Hamacher–Union with $\beta = 0.5$

Finally, with version (3) of the union function it is possible to submit a user defined co-norm.[4] For example, with

```
Hamacher_Union(Interval I, Interval J, Float beta)
= union(I, J, lambda(Float y1, Float y2)
   (y1+y2+((beta - 1)*y1*y2))/(1+beta*y1*y2))
```

one can define the same Hamacher union with a user defined co-norm.

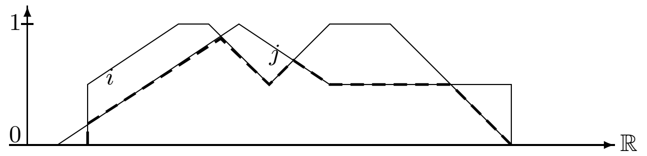
Definition 3.33 (Intersection of Intervals) *Let I and J be expressions of type* `Interval`, *The intersection operation for intervals comes also in three versions:*

(1) $\texttt{intersection}(I, J)$ $\qquad \texttt{Interval} * \texttt{Interval} \mapsto \texttt{Interval}$
(2) $\texttt{intersection}(I, J, \gamma)$ $\qquad \texttt{Interval} * \texttt{Interval} * \texttt{Float} \mapsto \texttt{Interval}$
(3) $\texttt{intersection}(I, J, norm))$
$\texttt{Interval} * \texttt{Interval} * (\texttt{Float} * \texttt{Float} \mapsto \texttt{Float}) \mapsto \texttt{Interval}$

■

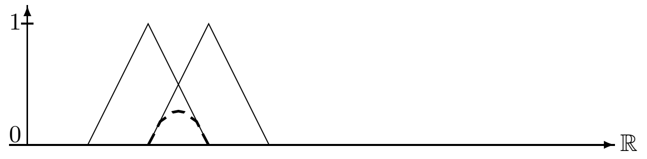
[4]Norms and co-norms are binary functions on membership values of fuzzy sets. They satisfy conditions which make sure that the corresponding set operations can be considered as union and intersection [5].

Version (1) is the standard intersection. Each pair (x, y_1) and (x, y_2) of points of the membership function of I and J is turned into $(x, min(y_1 - y_2))$.



Standard Intersection of Fuzzy Sets

Version (2) is the Hamacher–Intersection. For $\gamma \geq 0$, each pair (x, y_1) and (x, y_2) of points of the membership function of I and J is turned into $(x, \frac{y_1 y_2}{\gamma + (1-\gamma)(y_1 + y_2 - y_1 y_2)})$. The ordinary intersection is computed for $\gamma < 0$.



Hamacher–Intersection $\gamma = 0.5$

Finally, with version (3) it is possible to submit a user defined norm. For example, with

```
Hamacher_Intersection(Interval I, Interval J, Float gamma)
  = intersection(I, J, lambda(Float y1, Float y2)
          (y1*y2)/(gamma + (1-gamma)*(y1 + y2 -y1*y_2))
```

one can define the same Hamacher-Intersection with a user defined norm.

Definition 3.34 (Set Difference between Intervals) *Let I and J be expressions of type* `Interval`. *The set difference operation for intervals comes also in three versions:*

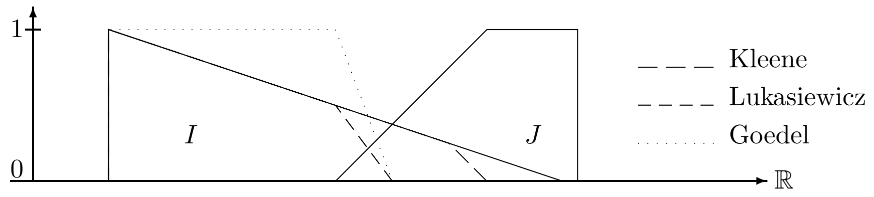
(1) `setdifference`(I, J) `Interval` $*$ `Interval` $\mapsto$ `Interval`
(2) `setdifference`$(I, J, version)$ `Interval` $*$ `Interval` $*$ `SDVersion` $\mapsto$ `Interval`
(3) `setdifference`$(I, J, intersection, complement)$
`Interval` $*$ `Interval` $*$ (`Interval` $*$ `Interval` $\mapsto$ `Interval`) $*$
(`Interval` $\mapsto$ `Interval`) $\mapsto$ `Interval`

■

(1) extends the crisp correspondence: $I \setminus J = I \cap J'$ where J' is the complement of J, `setdifference(I,J)` is therefore an abbreviation for `intersection(I,complement(J))` with standard intersection and complement functions.

(2) The second version computes the set difference operator by means of a binary function on the membership functions. The following versions are possible:

`SDVersion`	Function
`Kleene`	$(I \setminus J)(x) \stackrel{\text{def}}{=} min(I(x), 1 - J(x))$
`Lukasiewicz`	$(I \setminus J)(x) \stackrel{\text{def}}{=} max(0, I(x) - J(x))$
`Goedel`	$(I \setminus J)(x) \stackrel{\text{def}}{=} 0$ if $I(x) \leq J(x)$ and $1 - J(x)$ otherwise



Set Difference

(3) Finally, the third version is a generalization of the first version:

$$\texttt{setdifference}(I, J, \mathit{intersection}, \mathit{complement}) \overset{\text{def}}{=} \mathit{intersection}(I, \mathit{complement}(J))$$

intersection is a user defined binary function on intervals, and *complement* is a user defined unary function on intervals.

3.2.8 Predicates of Intervals

Fuzzy time intervals have many properties. They can be checked with suitable GeTS predicates.

Definition 3.35 (Predicates) *GeTS provides the following predicates to check the structure of an interval I:*

(1)	`isCrisp(`I`)`	`Interval` $\mapsto$ `Bool`
(2)	`isCrisp(`I`,left/right)`	`Interval` $*$ `Side` $\mapsto$ `Bool`
(3)	`isEmpty(I)`	`Interval` $\mapsto$ `Bool`
(4)	`isConvex(I)`	`Interval` $\mapsto$ `Bool`
(5)	`isMonotone(I)`	`Interval` $\mapsto$ `Bool`
(6)	`isInfinite(I)`	`Interval` $\mapsto$ `Bool`
(7)	`isInfinite(I,left/right)`	`Interval` $*$ `Side` $\mapsto$ `Bool`

■

`isCrisp`(I) checks whether the interval is a, possibly non-convex, crisp interval.
`isCrisp`(I, `left`) checks whether the interval is crisp at its left end. I may be infinite at this side, but the fuzzy value must be 1 in this case. Similar for `isCrisp`(I, `right`).

`isEmpty`(I) checks whether the interval is empty.

`isConvex`(I) checks whether the interval is convex. I can be non-convex even if I is crisp because it may consist of several different components (Def. 2.2).

`isMonotone`(I) checks whether the membership function of the interval is monotonically rising to a maximal value, and then monotonically falling again.

`isInfinite`(I) checks whether the interval is infinite.

`isInfinite`(I, `left`) checks whether the interval is infinite at the left hand side.
`isInfinite`(I, `right`) checks whether the interval is infinite at the right hand side.

The boundaries of infinite intervals are of course the infinity. Infinity has a special representation in the `Time` datatype. This can be checked with the `isInfinity` predicate:

Definition 3.36 (Infinity)

`isInfinity(time)`	`Time` $\mapsto$ `Bool`
`isInfinity(time,positive/negative)`	`Time` $*$ `PosNeg` $\mapsto$ `Bool`

■

`isInfinity(time)` checks whether the *time* represents an infinity.
`isInfinity(time,positive)` checks whether the *time* represents the positive infinity.
`isInfinity(time,negative)` checks whether the *time* represents the negative infinity.

The next three predicates allow one to check basic relations between time points and intervals, or between intervals and intervals.

Definition 3.37 (`during, isSubset, doesOverlap`)

(1)	`during`($time, I,$ `core/kernel/support`)	`Time` $*$ `Interval` $*$ `IntvRegion` $\mapsto$ `Bool`
(2)	`isSubset`($I, J,$ `core/kernel/support`)	`Interval` $*$ `Interval` $*$ `IntvRegion` $\mapsto$ `Bool`
(3)	`doesOverlap`($I, J,$ `core/kernel/support`)	`Interval` $*$ `Interval` $*$ `IntvRegion` $\mapsto$ `Bool`

■

(1) `during`($time, I, region$) checks whether the *time* is inside the given region of the interval I.
(2) `isSubset`($I, J, region$) checks whether the corresponding region of the interval I is a subset of the corresponding region of the interval J.
(3) `doesOverlap`($I, J, region$) checks whether the corresponding region of the interval I overlaps the corresponding region of the interval J.

The point-interval `during` relation is one of the five point–interval relations 'before', 'starts', 'during', 'finishes' and 'after' for crisp intervals. Only `during` is built-in because it is one of the most frequently used relations. The other relations can easily be defined in GeTS. The point–interval `before` relation in Example 3.3 is such an example.

3.2.9 Other Features of Intervals

With the first function in this paragraph one can access the fuzzy membership value of a time point within a given fuzzy interval.

Definition 3.38 (`member`) *The function*
`member`($time, I$)
of type
`Time *Interval` $\mapsto$ `Float`
returns the value of the membership function of the interval I at time point time. The value is a `Float` *number between 0 and 1.* ■

Definition 3.39 (Components)

1. *The function* `components`*(I) of type* `Interval` $\mapsto$ `Integer` *yields the number of components in the interval I.*

2. *The function* `component`*(I,k) of type* `Interval` $*$ `Integer` $\mapsto$ `Interval` *extracts the* $\mathtt{k}^{th}$ *component from the interval I.*

■

The function `size` below measures an interval I or parts of it by *integrating* over its membership function.

Definition 3.40 (`size`) *The function* `size` *comes in three versions.*

(1)	`size`(I)	`Interval` $\mapsto$ `Time`
(2)	`size`$(I,$ `core/support/kernel`$)$	`Interval` $*$ `IntvRegion` $\mapsto$ `Time`
(3)	`size`(I, t_1, t_2)	`Interval` $*$ `Time` $*$ `Time` $\mapsto$ `Time`

■

`size`(I) measures the size of the support of I.
`size`$(I,$ `core/support/kernel`$)$, measures the size of the corresponding region of I.
`size`$(I) =$ `size`$(I,$ `support`$)$.
`size`(I, t_1, t_2) measures the area of I between t_1 and t_2.

Definition 3.41 (sub and inf) *Let I be an interval expression.*
The function `sup`(I) *of type* `Interval` $\mapsto$ `Float` *returns the supremum of the fuzzy values of the membership function for I (usually 1).*

The function `inf`(I) *of type* `Interval` $\mapsto$ `Float` *returns the infimum of the fuzzy values of the membership function for I (usually 0).* ■

The function `length` measures the distance between two time points in terms of a partition.

Definition 3.42 (`length`) *The function*
`length`$(t_1, t_2, partitioning, asGranule)$
of type
`Time` $*$ `Time` $*$ `Partitioning` $*$ `Bool` $\mapsto$ `Float`
measures the distance between t_1 and t_2 in terms of the given partitioning. If asGranule = `true` *then the distance is measured in terms of the length of the granules of the partitioning's labelling (without gaps).* ■

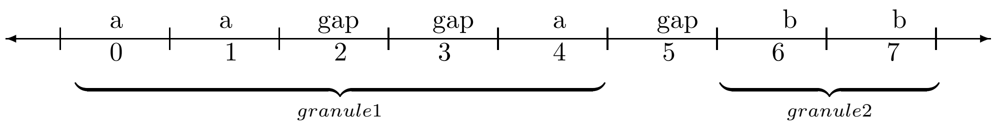
An example for determining the distance between two time points in terms of partitions is

```
length(now(),shift(now(),1 day),day,false)
```

is just 1.0.

The next example illustrates the `length` function in terms of granules.

Example 3.43 (`length` in terms of granules)
Consider a partitioning `P` with labelling a,a,gap,gap,a,gap,b,b.



The table below gives the results of `length(t1,t2,P,true)` where `t1` is the start of partition p1 and `t2` is the start of partition p2.

`t1`	`t2`	length in terms of granules
0	1	1/3
0	2	2/3
0	3	2/3
0	4	2/3
0	5	1
0	6	1
0	7	1.5
0	8	2

■

The function '`point`' below can be used to access the boundaries of the three different regions of an interval: support, core and kernel, and the first and last maximal points.

Definition 3.44 (`point`) *The function*

`point`(I, `left/right`, `core/support/kernel/maximum`)

of type

`Interval` $*$ `Side` $*$ `PointRegion` $\mapsto$ `Time`

returns the position of the boundaries of I's regions:

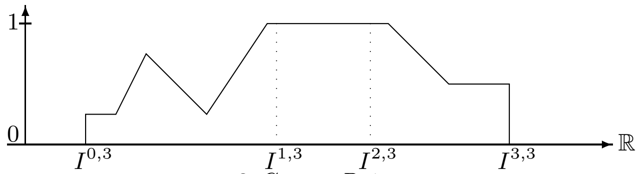
`point`(I,`left`,`support`)	*yields the position of the left support boundary*
`point`(I,`right`,`support`)	*yields the position of the right support boundary*
`point`(I,`left`,`core`)	*yields the position of the left core boundary*
`point`(I,`right`,`core`)	*yields the position of the right core boundary*
`point`(I,`left`,`kernel`)	*yields the position of the left kernel boundary*
`point`(I,`right`,`kernel`)	*yields the position of the right kernel boundary.*
`point`(I,`left`,`maximum`)	*yields the leftmost position of the maximal fuzzy value.*
`point`(I,`right`,`maximum`)	*yields the rightmost position of the maximal fuzzy value.*

■

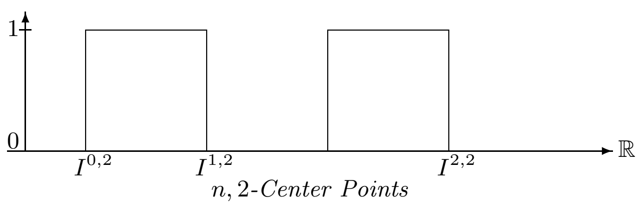
If `I` is just a convex crisp interval $[t_1, t_2[$ then
`point`(I,`left`,`support`) $= t_1$ and `point`(I,`right`,`support`) $= t_2$.

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

Example 3.45 (Center Points) *The 1,2-center point $I^{1,2}$ of I splits I in two halfs of the same size (integrated over the membership function). The 1,3-center point indicates a split of I into three parts of the same size.* `centerPoint(I,1,3)` *is the boundary of the first third,* `centerPoint(I,2,3)` *is the boundary of the second third.*



n, 3-Center Points



■

Definition 3.46 (Center Points) *The function*
`centerPoint`(I, n, m)
of type
`Interval` $*$ `Integer` $*$ `Integer` $\mapsto$ `Time`
yields the (earliest) position of the n, m-center point. ■

The center points are computed such that for $n < m$:

$$\int_{centerPoint(n,m)}^{centerPoint(n+1,m)} I(x)\ dx = (\int I(x)\ dx)/m$$

3.2.10 Basic Manipulations of Intervals

In this paragraph we introduce some elementary transformation functions for fuzzy time intervals.

Definition 3.47 (Shift of Time Intervals) *The function*
`shift`(I, t)
of type
`Interval` $*$ `Time` $\mapsto$ `Interval`
shifts the interval by the given time, i.e. `shift`$(I, t)(x) = I(x - t)$ ■

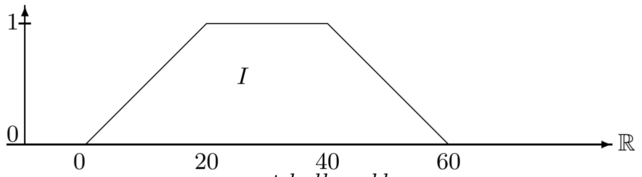
Definition 3.48 (`cut`) *The function*
`cut`(I, t_1, t_2)
of type
`Interval` $*$ `Time` $*$ `Time` $\mapsto$ `Interval`
cuts the part of the interval I between the time points t_1 and t_2 out of I and returns it as a new interval. ■

The `hull` function below is able to compute different hulls of a fuzzy time intervals.

Definition 3.49 (Hull Calculations) *The function*
`hull`$(I,$ `core/support/kernel/crisp/monotone/convex`$)$
of type
`Interval` $*$ `Hull` $\mapsto$ `Interval`
computes a hull of the interval I. The second parameter determines which hull is to be computed. ■

The `core`, `support` and `kernel` hull compute the corresponding interval regions as crisp intervals. The `core` and `support` hull may therefore consist of different components, whereas the `kernel` hull consists of at most one single component.

There is a small problem with the `support` hull. Consider the following example:



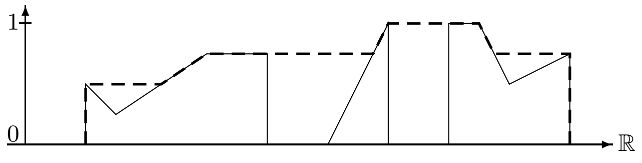
support hull problem

Since $I(0) = 0$, the support of I is the open interval $]0, 60[$. The function `hull(I,support)`, however, calculates the interval boundaries 0 and 60, which are interpreted as the half open interval $[0, 60[$. Strictly mathematical, this is not correct. In a correct implementation, however, we would have to distinguish open and half open intervals. Since the overhead for this is enormous, the current version of GeTS has to live with this error.

The `crisp` hull for crisp intervals is the usual convex hull of crisp intervals. It consists of the smallest crisp interval which contains all the components of the interval. The `crisp` hull for non-crisp intervals is the convex hull of the support of the interval. If the non-convex interval consists of one single component only, there is no difference between the crisp and support hull. In general we have

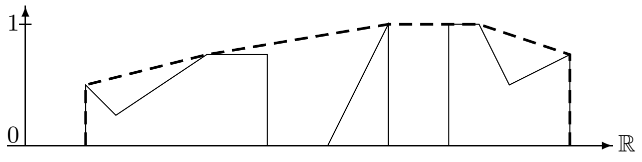
$$\texttt{hull}(I, \texttt{crisp}) = \texttt{hull(hull}(I, \texttt{support),crisp)}.$$

The `monotone` hull of an interval I is the smallest *monotone* interval which contains I. An interval is *monotone* iff its membership function rises monotonically up to a maximal point, and then falls monotonically again.



Monotone Hull of a Fuzzy Interval

The `convex` hull of an interval I is the smallest *convex* interval which contains I. The notion 'convex', which is appropriate here, is the notion of a *convex polygon.* That means, if we follow the membership function from left to right there are only right curves. The next figure illustrates this.



Convex Hull of a Fuzzy Interval

If the interval I is crisp then the crisp, monotone and convex hull are the same.

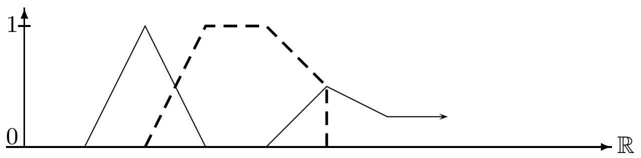
The next function can be used to extract the gaps between components of an interval. The `invert` function inverts the membership function, but only between the last maximal point of the first component and the first maximal point of the last component. `invert(I)` is zero outside these points.

Definition 3.50 (`invert`) *The function* `invert`(I) *of type* `Interval` $\mapsto$ `Interval` *inverts the membership function of the interval* I*:*

$$\texttt{invert}(I)(x) \overset{\text{def}}{=} \begin{cases} 1 - I(x) & \text{if } a \leq x < b \\ 0 & \text{otherwise.} \end{cases}$$

where a *is the last maximal point of the first component of* I*, and* b *is the first maximal point of the last component of* I*.* ■

Example:



Invert

`components`$($`invert`$(I))$ yields the number of gaps in the interval I.

The `scaleup` function below multiplies the membership function of an interval I with a factor f, such that the maximal value of $I(x) * f$ is 1.

Definition 3.51 (`scaleup`) *The function* `scaleup`(I) *of type* `Interval` $\mapsto$ `Interval` *scales the membership function of* I *such that its maximum is 1.* ■

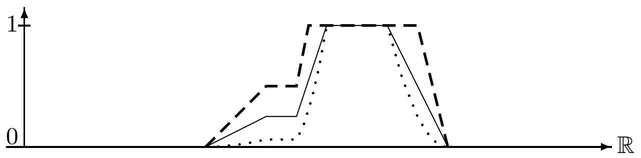
More general scaling functions are `times` and `exp`.

Definition 3.52 (`times` and `exp`)

$$\begin{array}{ll} \texttt{times}(I, f) & \texttt{Interval} * \texttt{Float} \mapsto \texttt{Interval} \\ \texttt{exp}(I, e) & \texttt{Interval} * \texttt{Float} \mapsto \texttt{Interval} \end{array}$$

`times`$(I, f)(x) = min(I(x) \cdot f, 1)$.
`exp`(I, e) *computes an interval such that* $exp(I, e)(x) = I(x)^e$. ■

The dashed line in the next figure indicates `times`$(I, 2)$ and the dotted line indicates `exp`$(I, 2)$.



`times`$(I, 2)$ *and* `exp`$(I, 2)$

The rising part of a fuzzy time interval is crucial for a fuzzy point-interval `before` relation. The falling part, on the other hand, is crucial for a point-interval `after` relation. The rising part of an interval I can be computed by following its monotone hull up to the first maximal point, and then extending it to the infinity. Similar with the falling part.

Definition 3.53 (Extend to Infinity) *The function*

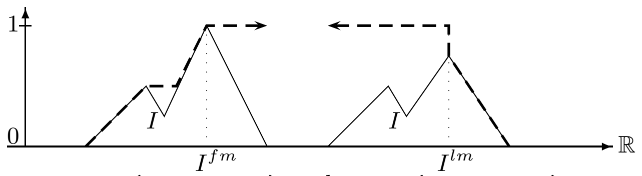
`extend`$(I,$`positive/negative`$)$

of type

`Interval` $*$ `PosNeg` $\mapsto$ `Interval`

extends the interval to the infinity. `extend`$(I,$`positive`$)$ *raises the membership function of the monotone hull of I to 1 after the first maximum I^{fm}.* `extend`$(I,$`negative`$)$ *raises the membership function of the monotone hull of I to 1 before the right maximum I^{lm}.* ■

Example:

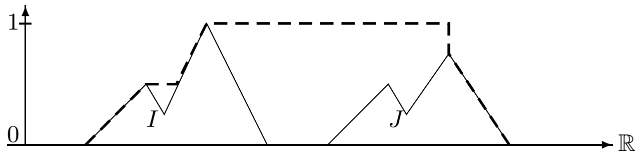


`extend(I,positive)` *and* `extend(I,negative)`

An example where the `extend` function is useful is the definition of the binary 'until' relation between two intervals.

```
until(Interval I, Interval J)
= intersection(extend(I,positive),extend(J,negative))        (3)
```

computes `until`(I, J) as the interval which lasts from the beginning of interval I until the end of interval J.



until

There is a further `extend` function in GeTS. It lengthens or shortens an interval by a certain time.

Definition 3.54 (Extend by a Certain Time) *The function*

`extend`$(I, \textit{length}, \textit{side})$

of type

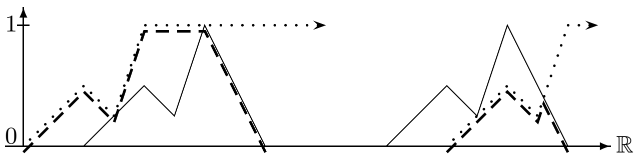
`Interval` $*$ `Time` $*$ `Side` $\mapsto$ `Interval`

extends the interval I by the given length. ■

The *side* parameter determines at which side the interval is extended. *side* = `left` extends it at the left side, *side* = `right` extends it at the right side. A positive *length* value causes the interval to be extended, whereas a negative *length* value causes the interval to be shrunken.

The algorithm for extending or shrinking a fuzzy interval works as follows: In a first step the interval I is split into the left/right part I_1 of the interval up to the first maximal point, and the rest I_2. I_1 is extended to the infinity. This part is shifted. If the interval is to be extended, then the union of the shifted I_1 with I_2 is computed. If the interval is to be shrinked then the intersection of the shifted I_1 with I_2 is computed. The next figure illustrates this. The dotted line shows the shifted front part of the interval. The dashed line is the result of the union/intersection.

Example:



extending and shrinking an interval by a certain duration

The `extend` function together with `shiftLength` (Def. 3.28) can be used to extend an interval by a certain *duration*. For example,

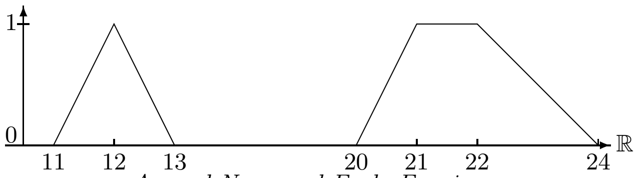
$$\texttt{extend}(I, -\texttt{shiftLength}(\texttt{point}(I, \texttt{left}, \texttt{support}), -1\ \texttt{month}, \texttt{false}, \texttt{true}), \texttt{left})$$

extends the left side of the interval I by 1 month. The month length is determined by a backwards shift of the left boundary of I's support.

Definition 3.55 (`integrate`) *The function*
`integrate`$(I,$`positive/negative`$)$
of type
`Interval` $*$ `PosNeg` $\mapsto$ `Interval`
integrates the membership function of I and normalizes its value to 1. If the control parameter is `positive` *then I is integrated from left to right. If it is* `negative` *then I is integrated from right to left.* ■

An example where the `integrate` operator may be useful is the definition of *party time*

Example 3.56 (Birthday Party Time) Consider a database about, say, the institute's birthday parties. It may contain the entry that the birthday party for the director took place 'from around noon until early evening' of 20/7/2003. 'Around noon' is a fuzzy notion and 'early evening' is a fuzzy notion. Suppose, we have a formalization of 'around noon' and 'early evening' as the following fuzzy sets:

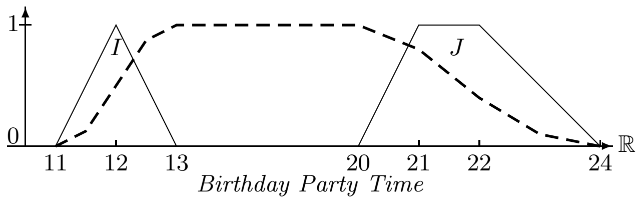


Around Noon and Early Evening

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

```
partyTime(Interval I, Interval J)
= intersection(integrate(I,positive),integrate(J,negative))
```
(4)

The resulting fuzzy set is:



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

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

Definition 3.57 (Fuzzification) *There are two different versions of the* `fuzzify` *function in GeTS. The first version allows one to specify the part of the interval I which is to be fuzzified in terms of percents of the interval length. The second version needs absolute coordinates.*

`fuzzify`(I, `linear/gaussian`, `left/right`, *increase*, *offset*)
of type
`Interval`, `Fuzzify`, `Side`, `Float`, `Float` $\mapsto$ `Interval`

`fuzzify`(I, `linear/gaussian`, `left/right`, $x1$, $x2$, *offset*)
of type
`Interval`, `Fuzzify`, `Side`, `Time`, `Time`, `Time` $\mapsto$ `Interval`

■

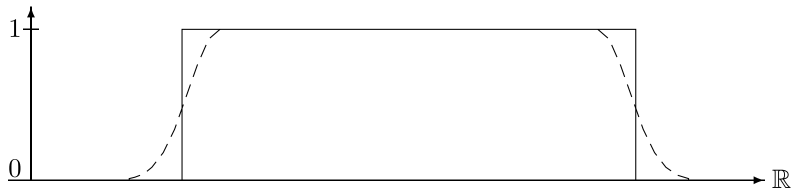
The second parameter determines whether a linear or gaussian increase is to be imposed on the interval. The third parameter determines whether the increase is from left to right or from right to left. *increase* is a Float number in percent. *increase* = 10 means that the region to be modified consists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

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

A function which fuzzifies both ends of an interval in the same way could be

```
f(Interval I, Float increase, Float offset)
  = intersection(extend(fuzzify(I,gaussian,left,increase,offset),positive),
                 extend(fuzzify(I,gaussian,right,increase,offset),negative))
```

$f(I, 20, 0)$ produces the following fuzzified interval.



Relative Gaussian Fuzzification

Notice that the obvious 'solution'

```
f(Interval I, Float increase, Float offset)
  = fuzzify(fuzzify(I,gaussian,right,inc,off),left,increae,offset)
```

yields no symmetric structure, because the inner `fuzzify` operation changes the kernel of the interval, such that the absolute increase and offset of the outer `fuzzify` operation are different to the absolute increase and offset of the inner `fuzzify` operation

The next example illustrate a potential use of the `fuzzify` function. We want to realize a function `beforeChristmas`. It should accept a time point t and compute a fuzzy interval, whose membership function increases for a certain time period and then stays 1.0 until Christmas. The increase is determined by two parameters, `offset` and `increase`. `offset` $= 50$ means that the increase should start in the middle between t and Christmas. `increase` $= 20$ means that the duration of the actual linear increase should be 20% of the interval length.

If $t = 2004/7/1$ then `beforeChristmas(t,50,10)` yields an interval whose membership function rises from 2004/9/28 until 2004/10/6/19/12 and then stays at 1.0 until 2004/12/25.

Example 3.58 (Before Christmas)

```
beforeChristmas(Time t, Float offset, Float increase) =
   dLet year = date(t,Gregorian_month) in
      Let christmas = time(year|12|25,Gregorian_month) in
          case (t < christmas) :
            Let days = round(length(t,christmas,day,false),down) in
                 fuzzify([time(year|12|25-days+round((days*offset/100)),
                               Gregorian_month),christmas],
                        linear,left,increase,0),
          (t < time(year|12|27,Gregorian_month)): []
          else
             Let christmas1 = time(year+1|12|25,Gregorian_month) in
              Let days = round(length(t,christmas1,day,false),down) in
                  fuzzify([time(year+1|12|25-days+round((days*offset/100)),
                                Gregorian_month),christmas1],
                            linear,left,increase,0)
```

■

The `beforeChristmas` function considers the three cases, namely (1) that the time point t in the year y is before Christmas in this year, (2) that t is just on Christmas in this year, and

(3) that t is after Christmas in this year. In case (1) the rounded number of days between t and Christmas is computed first (line 5). This number minus the offset is subtracted from `christmas` to get the left boundary of the interval to be fuzzified (line 6). The right boundary is `christmas`. The left part of the interval is fuzzified linearly with the given increase (line 6–8). If the time point t is just on Christmas (line 9) then the empty interval is returned. If t is after Christmas (case 3), then next year's Christmas is considered (line 11-15).

Integration over Pairs of Intervals
One possibility to define an interval–interval relation like '$before(I, J)$' is, to take a point–interval relation '$PIRbefore(t, J)$' and average $PIRbefore(t, J)$ over the interval I. Averaging over an interval means integrating over its membership function. For purposes like this GeTS provides two integration operations.

Definition 3.59 (Integration) *GeTS has the two integration functions:*

$$\begin{array}{ll} \texttt{integrateSymmetric}(I, J, simple) & \texttt{Interval} * \texttt{Interval} * \texttt{Bool} \mapsto \texttt{Float} \textit{ and} \\ \texttt{integrateAsymmetric}(I, J) & \texttt{Interval} * \texttt{Interval} \mapsto \texttt{Float} \end{array}$$

■

$\texttt{integrateAsymmetric}(I, J)$ computes $(\int I(x) \cdot J(x)\ dx)/|I|$.

$\texttt{integrateSymmetric}(I, J, simple)$ computes $(\int I(x) \cdot J(x)\ dx)/N(I, J)$

$$\text{where } N(I, J) \overset{\text{def}}{=} \begin{cases} min(|I|, |J|) & \text{if } simple = \texttt{true} \\ max_a(\int I(x-a) \cdot J(x)\ dx) & \text{otherwise.} \end{cases}$$

Example 3.4 shows an application of the asymmetric `integrate` function.

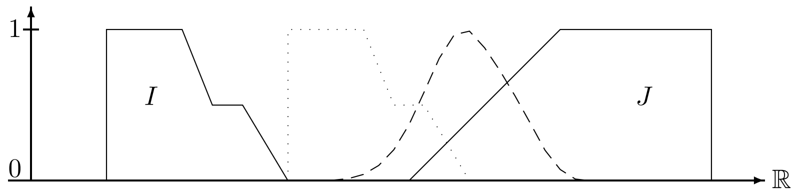
The next example shows an application of the symmetric `integrate` function. A fuzzy interval–interval relation `IIRMeets` is defined: Besides the two intervals, it takes the transformation functions F and S and integrates the interval $F(I)$ over $S(J)$. $F(I)$ should map the interval I to a finishing section of I and $S(J)$ should map the interval J to a starting section of J. The integration of $F(I)$ to $S(J)$ yields the final result.

Example 3.60 (Fuzzy Interval–Interval 'Meets' Relation) A possible definition for a fuzzy interval–interval meets relation is

```
IIRMeets(Interval I, Interval J, Interval->Interval F, Interval->Interval S) =
  if isEmpty(I) or isEmpty(J) or isInfinite(I,right) or isInfinite(J,left)
    then 0
  else integrateSymmetric(F(I),S(J),false)
```

■

The figure below shows the effect of the `IIRMeets` relation for suitable `F` and `S` operations. The dashed figure shows the result of $\texttt{IIRMeets}(I, J, \ldots)$ when the interval I is moved along the time axis. The dotted figure shows the position of the interval I where $\texttt{IIRMeets}(I, J, \ldots)$ is maximal.



`IIRMeets` *for Fuzzy Intervals*

GeTS contains the very special purpose function `MaximizeOverlap` which is, so far, only needed for implementing the fuzzy interval–interval overlaps relation. The classical relation I *overlaps* J has two requirements:
1. a non-empty part I_1 of I must lie before J, and
2. another non-empty part I_2 of I must lie inside J.

A generalization to fuzzy intervals encodes the first condition in the factor $1 - D(I, E^+(J))$ where D is a `during` operator. $E^+(J)$ extends the rising part of J to the infinity. Therefore $D(I, E^+(J))$ measures the part of I which is after the front part of J. $1 - D(I, E^+(J))$ then measures the part of I which is before the front part of J. This factor is multiplied with $D(I, J)$ which corresponds to the second condition. It measures to which degree I is contained in J. The product is normalized with $\max_a((1 - D(I_a, E^+(J))) \cdot D(I_a, J))$, where $I_a(x) \overset{\text{def}}{=} I(x - a)$. This corresponds to the maximal possible overlap when I is shifted along the time axis. This guarantees that there is a position for I where I *overlaps* $J = 1$. The normalization factor is computed with the function `MaximizeOverlap`

Definition 3.61 (`MaximizeOverlap`) *The function*
`MaximizeOverlap`(I, J, EJ, D)
of type
`Interval` $*$ `Interval` $*$ `Interval` $*$ (`Interval` $*$ `Interval` $\mapsto$ `Float`) $\mapsto$ `Float`
computes

$$\max_a((1 - D(\mathtt{shift}(I, a), EJ)) \cdot D(\mathtt{shift}(I, a), J))$$

■

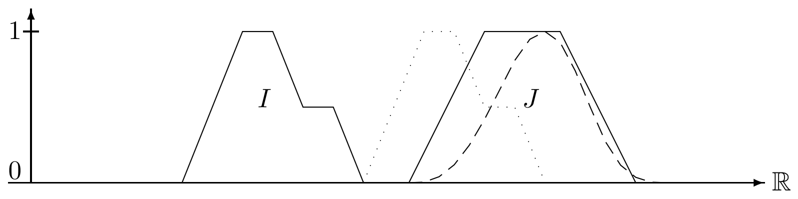
Notice that EJ can in principle be an arbitrary interval. For the encoding of the fuzzy overlaps relation, it should, however, be the extension of J to the infinity.

Example 3.62 *IIROverlaps*

```
IIROverlaps(Interval I, Interval J, Interval->Interval E,
            (Interval*Interval)->Float D) =
  case
    isEmpty(I) or isEmpty(J) or isInfinite(J,left) : 0,
    isInfinite(I,right) : float(point(I,left, support)  < point(J,left, support)),
    isInfinite(J,right) : float(point(I,right, support) < point(J,left, support))
  else
    Let EJ = E(J) in
       (1 - D(I,EJ))*D(I,J) / MaximizeOverlap(I,J,EJ,D)
```

■

Example 3.63 (IIROverlaps for Fuzzy Intervals)
This example shows the result of the `IIROverlaps` relation where the standard `IIRDuring` operator is used (with the identity function as point–interval `during` operator).



Example: Overlaps Relation

The dashed line represents the result of the overlaps relation for a time point t where the positive end of the interval I is moved to t. The dotted figure indicates the interval I moved to the position where `IIRoverlaps`(I, J) becomes maximal. ■

3.2.11 Date and Time

In examples 3.2 and 3.58 we have already seen applications of functions which convert time points to dates and dates to time points. The dates are sequences of integers which correspond to date formats, and these are sequences of partitionings. An example for a date format is year/month/day/hour/minute/second in the Gregorian calendar. The sequence 2004|12|3|21|43|0 in this date format is therefore the 3rd of December 2004, 9:43 pm.

The `time` function converts a date in a given date format to the corresponding time point.

Definition 3.64 (`time`) *The function*
`time`$(year|month|..., dateFormat)$
of type
`Integer` $*\ldots*$ `Integer` $*$ `DateFormat` $\mapsto$ `Time`
maps a date in a given date format to the time point denoted by this date. ■

The tokens *year*, *month* etc. in the `time` function are expressions of type `Integer`. There can be as many expressions as the date format has partitionings. For example, the year/month/day/hour/minute/second date format in the Gregorian calendar has 6 partitionings. Therefore there are in this case at most 6 `Integer` expressions allowed in the `time` function.

Examples:
`time(2004,Gregorian_month)` = 1072915231 (1st of January 2004)
`time(2004|1+1,Gregorian_month)` = 1075593631 (1st of February 2004)
`time(2004|2|2,Gregorian_week)` = 1073347231 (6th of January 2004)
`Gregorian_week` is the date format year/week/day/hour/minute/second. Therefore 2004|2|2 is the second day in the second week in the year 2004.

The `dLet` construct in the next definition is a kind of inverse to the `time` function. It computes for a given time point and a date format a date representation as a sequence of `Integer`s and binds the variables to these `Integer`s in a similar way as the `Let` construct.

Definition 3.65 (`dLet`) *The expression*

$$\texttt{dLet}\ year|month|... = \texttt{date}(time, dateFormat)\ \texttt{in}\ expression$$

binds the variables $year, month, \ldots$ to the integers which correspond to the date denoted by 'time', in the given date format.
'expression' is then evaluated under this binding.
The type of `date` *is* $\texttt{Time}{*}\texttt{DateFormat} \mapsto \texttt{Integer}^n$ *where* $n \leq$ *maximal number of partitionings in the date format.*

■

Example:
'$\texttt{dLet}\ y|m|d|h = \texttt{date}(0, \texttt{Gregorian\_month})\ \texttt{in}\ y + m + d$' yields 1973 because the time point 0 corresponds to the first of January 1970. Therefore $y = 1970$, $m = 1$, $d = 1$ and $h = 0$.

3.2.12 Partitionings and Labels

GeTS has a number of functions for reckoning with time points, partitions and labels. The `partition` function was already introduced in Example 3.1.

Definition 3.66 (`partition`) *The* `partition` *function maps time points to intervals, which represent partitions.*

(1) $\texttt{partition}(time, partitioning)$
  $\texttt{Time} * \texttt{Partitioning} \mapsto \texttt{Interval}$

(2) $\texttt{partition}(time, partitioning, n, m)$
  $\texttt{Time} * \texttt{Partitioning} * \texttt{Integer} * \texttt{Integer} \mapsto \texttt{Interval}$

■

The first version computes the interval which corresponds to the partition containing *time*. The second version computes an interval $[t_1, t_2[$ as follows: If i is the coordinate of the partition containing *time* then t_1 is the start of partition $i + n$ and t_2 is the end of the partition $i + m$.

If instead of the partition as interval, only the boundaries are needed, one can use the `partitionBoundary` function.

Definition 3.67 (partition boundary) *The function*
  $\texttt{partitionBoundary}(time, partitioning, \texttt{left/right})$
  of type
  $\texttt{Time} * \texttt{Partitioning} * \texttt{Side} \mapsto \texttt{Time}$
computes the left/right boundary of the partition containing time. ■

Although partitionings are in general infinite mathematical structures, their validity may be limited (see remark 2.5). GeTS has two functions for getting information about the boundaries of the valid regions of a partitioning.

Definition 3.68 (Valid Regions of Partitionings) *The function*
  $\texttt{partitioningIsBounded}(P, \texttt{left/right})$
  of type
  $\texttt{Partitioning} * \texttt{Side} \mapsto \texttt{Bool}$
checks whether the valid region of the partitioning P is bounded at the given side.

The function
  $\texttt{partitioningBoundary}(P, \texttt{left/right})$
  of type
  $\texttt{Partitioning} * \texttt{Side} \mapsto \texttt{Time}$

returns the boundary of the partitioning at the given side. If there is no bound at this side then a representation of infinity is returned. ∎

The next function is `which`. It can, for example, be used to compute which week in the year is now, or which day in the semester is now.

Definition 3.69 (`which`) *The function*

`which`$(\mathit{time}, P, Q, \mathit{inclusion}, \mathit{asGranule})$

is of type

$\texttt{Time} * \texttt{Partitioning} * \texttt{Partitioning} * \texttt{Inclusion} * \texttt{Bool} \mapsto \texttt{Time}$. ∎

The function is explained for the two *asGranule* case:

Case $\mathit{asGranule} = \texttt{false}$**:**
Consider the following example:
`which(now(),week,year,bigger_part_inside,false)`.
It computes, which week of the year is now.

The `which` function first computes the starting point t_Q of the Q-partition containing *time*. In the example, it would be the beginning of the current year. Then it determines the P-partition for t_Q. In the example, it is the first week in the year. What counts as the 'first' P-partition p depends on the parameter *inclusion*:

$\mathit{inclusion} = \texttt{subset}$**:** p is the leftmost P-partition after t_Q.

$\mathit{inclusion} = \texttt{overlaps}$**:** p is the leftmost P-partition containing t_Q.

$\mathit{inclusion} = \texttt{bigger\_part\_inside}$**:** p is the leftmost P-partition whose bigger part comes after t_Q. (This is suitable for counting weeks within a year).

If n is the coordinate of p and m is the coordinate of the P-partition containing *time* then $m - n$ is returned by the `which` function. If `now()` is first of January then the call
`which(now(), week, year, bigger_part_inside, false)`
returns 0 as the number of the first week in the year.

Notice that the result of the `which` function is of type `Time`. The `Time` datatype is in this case just to be taken as a potentially very big integer, and not as a time point.

Case $\mathit{asGranule} = \texttt{true}$**:**
The partitionings P and Q are in this case interpreted as granules. t_Q is the start of the Q-granule containing *time*. If *time* is between two granules then t_Q is the end of the Q-granule before t_Q.

The first P granule p depends again on the parameter *inclusion*:

$\mathit{inclusion} = \texttt{subset}$**:** p is the leftmost P-granule after t_Q.

$\mathit{inclusion} = \texttt{overlaps}$**:** p is the leftmost P-granule containing t_Q. If t_Q is between two P-granules then p is the leftmost P-granule after t_Q.

$\mathit{inclusion} = \texttt{bigger\_part\_inside}$**:** p is the leftmost P-granule whose bigger part comes after t_Q.

A special case is that *time* lies before the start of p. In this case the `which` function returns the value -1 to indicate that the counting is not possible.

In the normal case the `which` function counts from granule p as number 0 the P-granules until it reaches *time*. If *time* is between two P-granules then the counting stops before *time*. The value of the counter is returned.

The functions in the next definition deal with labels of partitions. Notice that labels are not just strings. They are special data structures, such that, for example, two labels with the same name are identical.

Definition 3.70 (Basic Functions for Labels) *The function*

`label`$(time, partitioning)$

of type

`Time` $*$ `Partitioning` $\mapsto$ `Label`

returns the label of the partition containing time. If there is no labelling defined, it returns a NULL label.

The function

`isLabel`$(label)$

of type

`Label` $\mapsto$ `Bool`

checks whether the label is not the NULL label.

The function

`isGap`$(label)$

of type

`Label` $\mapsto$ `Bool`

checks whether the label is the gap label.

The function

`LabelName`$(name)$

of type

`String` $\mapsto$ `Label`

turns a string (without quotes) into a `Label`*.* ■

The `extractLabelled` function below can be used to extract from an interval all partitions with a given label, for example all Tuesdays of a labelled day partitioning.

Definition 3.71 (`extractLabelled`) *The function*

`extractLabelled`$(I, label, partitioning, inclusion, intersect)$

of type

`Interval` $*$ `Label` $*$ `Partitioning` $*$ `SplitInclusion` $*$ `Bool` $\mapsto$ `Interval`

extracts partitions in the interval I with the given label. ■

The `extractLabelled` function maps through all partitions of the given partitioning which are labelled with the given label, and which overlap with the interval $[a, b[$ where a is the left boundary of the interval and b is the right boundary of the interval. An error is thrown if a or b are the infinity.

For each such partition p a condition is tested which depends on the parameter *inclusion*.

inclusion $=$ **`align`:** the condition is always true.

$inclusion =$ `subset`: p must be a subset of I's support.

$inclusion =$ `overlaps`: p must overlap with I's support.

$inclusion =$ `bigger_part_inside`: the bigger part of p must be a subset of I's support.

If the parameter $intersect =$ `false` then all partitions p which meet the condition are joined into the resulting (crisp) interval.

If the parameter $intersect =$ `true` then the intersection of I with all partitions p which meet the condition are joined into the resulting interval. The result may now be a fuzzy interval.

The function below is for constructing intervals which represent granules.

Definition 3.72 (`nextGranule`) *The function*

`nextGranule`$(time, partitioning, label, n, withGaps)$
of type
$\texttt{Time} * \texttt{Partitioning} * \texttt{Label} * \texttt{Integer} * \texttt{Bool} \mapsto \texttt{Interval}$

constructs a new interval which represents a granule. ■

The interval is constructed as follows:
Case 1: *time* is inside a granule with the given *label*.
If $n = 0$ then this granule is computed. Otherwise the n^{th} next/previous (if $n < 0$) granule with this label is computed.
Case 2: *time* lies outside a granule with the given *label*.
If $n = 0$ then the empty interval is returned. Otherwise the n^{th} next/previous (if $n < 0$) granule with this label is computed.

Finally an interval is constructed and returned which represents the granule. If $withGaps =$ `true` then this interval may be non-convex to exclude the gap partitions within a granule.

3.2.13 Control Constructs for Operations on Intervals

GeTS has two basic control constructs for operations on parts of intervals. The `componentwise` control construct allows one to apply an operation to each component of an interval and to combine the results of each application with a combination function.

Definition 3.73 (`componentwise`) *The following function applies an operation to each component of an interval and combines the results with a combination operator. It comes in two versions, without and with an end test.*

`componentwise`$(I, initialObject, operation, combination)$
of type
$[\texttt{Interval} * T * (\texttt{Interval} \mapsto T) * (T * T \mapsto T) \mapsto T]$

`componentwise`$(I, initialObject, operation, combination, endTest)$
of type
$[\texttt{Interval} * T * (\texttt{Interval} \mapsto T) * (T * T \mapsto T) * (T \mapsto \texttt{Bool}) \mapsto T]$

■

I is the interval whose components are considered.
operation is the operation which is applied to the components of I. It generates results of type T (which is determined by the type of *initialObject*).
combination is the operation which is used to combine the results of the application of *operation*.
initialObject is the object which is returned when I is empty, and which is used to combine it with the very first result of *operation*. Typically, $initialObject = []$ ($T =$ `Interval`) or $initialObject = 0T$ ($T =$ `Time`).
endTest is a predicate which is applied to the intermediate results. The loop is terminated and the intermediate result is returned as soon as *endTest* yields `true`.

Examples:
$\texttt{componentwise}(I, [], \texttt{lambda(Interval } J\texttt{) hull}(J, \texttt{crisp}),$
$\qquad\qquad\qquad\qquad \texttt{lambda(Interval } K, \texttt{Interval } L\texttt{) union}(K, L))$
computes the crisp hull for each component of the interval I separately and then joins them into one single crisp, possibly non-convex, interval.

$\texttt{componentwise}(I, 0.0,$
$\quad \texttt{lambda(Interval } J\texttt{) length(point}(J, \texttt{left}, \texttt{support}),$
$\qquad\qquad\qquad\qquad\qquad \texttt{point}(J, \texttt{right}, \texttt{support}), \texttt{month}, \texttt{false})$
$\quad \texttt{lambda(Float } n, \texttt{Float } m\texttt{) } n + m)$
computes the lengths of the support of (a finite) interval I in terms of months.

split:
The `split` function below is in principle similar to the `componentwise` function. The difference is that the interval is not taken apart into its components, but it is split into subintervals of a given length. A function is applied to these split parts, and a combination function combines the partial results into a final result.

Definition 3.74 (`split`) *The* `split` *function also comes in two versions, without and with an end test.*
$\texttt{split}(I, duration, asGranule, dateOriented, initialObject, operation,$
$\quad combination, region, forward, inclusion, sequencing, intersect)$
is of type
$\texttt{Interval} * \texttt{Duration} * \texttt{Bool} * \texttt{Bool} * T * (\texttt{Interval} \mapsto T) * (\texttt{Interval} * \texttt{Interval} \mapsto T)*$
$\quad \texttt{IntvRegion} * \texttt{Bool} * \texttt{SplitInclusion} * \texttt{Sequencing} * \texttt{Bool} \mapsto T$

$\texttt{split}(I, duration, asGranule, dateOriented, initialObject, operation,$
$\quad combination, region, forward, inclusion, sequencing, intersect, endTest)$
is of type
$\texttt{Interval} * \texttt{Duration} * \texttt{Bool} * \texttt{Bool} * T * (\texttt{Interval} \mapsto T) * (\texttt{Interval} * \texttt{Interval} \mapsto T)*$
$\quad \texttt{IntvRegion} * \texttt{Bool} * \texttt{SplitInclusion} * \texttt{Sequencing} * \texttt{Bool} * (T \mapsto \texttt{Bool}) \mapsto T.$
■

In order to explain the `split` function in detail, we must introduce some auxiliary functions. They are not part of GeTS, but used internally.
The *startpoint* and *endpoint* functions compute the starting point for the split.

Definition 3.75 (*startpoint* and *endpoint*) The function $startpoint(t, P, inclusion, asGranule)$ of type `Time` $*$ `Partitioning` $*$ `SplitInclusion` $*$ `Bool` $\mapsto$ `Time` computes the starting point of a forward split as follows:

Case *asGranule* = `false`**:**

- **Case** *inclusion* = `align`**:** return t
- **Case** *inclusion* = `subset`**:** return the starting point s of the leftmost P-partition such that $t \leq s$.
- **Case** *inclusion* = `overlaps`**:** return the starting point of the leftmost P-partition containing t.
- **Case** *inclusion* = `bigger_part_inside`**:** return the starting point of the leftmost P-partition p such that the bigger part of p comes after t.

Case *asGranule* = `true`**:**

- **Case** *inclusion* = `align`**:** If t is between two granules g_1 and g_2 then return the starting point of g_2, otherwise return t.
- **Case** *inclusion* = `subset`**:** return the starting point of the leftmost granule after t.
- **Case** *inclusion* = `overlaps`**:** If t is between two granules g_1 and g_2 then return the starting point of g_2, otherwise return the starting point of the leftmost granule containing t.
- **Case** *inclusion* = `bigger_part_inside`**:** If t is between two granules g_1 and g_2 then return the starting point of g_2, otherwise return the starting point of the leftmost granule whose bigger part comes after t. Gaps within a granule are not measured.

A similar function $\mathit{endpoint}(t, P, \mathit{inclusion}, \mathit{asGranule})$ computes the starting point of a backwards split:

Case *asGranule* = `false`**:**

- **Case** *inclusion* = `align`**:** return t
- **Case** *inclusion* = `subset`**:** return the end point s of the rightmost P-partition such that $s \leq t$.
- **Case** *inclusion* = `overlaps`**:** return the endpoint of the rightmost P-partition containing t.
- **Case** *inclusion* = `bigger_part_inside`**:** return the endpoint of the rightmost P-partition p such that the bigger part of p comes before t.

Case *asGranule* = `true`**:**

- **Case** *inclusion* = `align`**:** If t is between two granules then return the endpoint of the rightmost granule before t, otherwise return t.
- **Case** *inclusion* = `subset`**:** return the endpoint of the rightmost granule before t.
- **Case** *inclusion* = `overlaps`**:** If t is between two granules then return the endpoint of the rightmost granule before t, otherwise return the endpoint of the rightmost granule containing t.

Case *inclusion* = `bigger_part_inside`**:** If t is between two granules then return the endpoint of the rightmost granule before t, otherwise return the endpoint of the rightmost granule whose bigger part comes before t. Gaps within a granule are not measured.

■

The functions *advance* and *retract* below compute the start of the next split part.

Definition 3.76 (*advance* **and** *retract***)** The function $advance(t, P, sequencing, asGranule)$ of type `Time` $*$ `Partitioning` $*$ `Sequencing` $*$ `Bool` $\mapsto$ `Time` computes for the end time t of a (forward) split part the start time of the next split part:

Case *asGranule* = `false`**:**

Case *sequencing* = `sequential`**:** return t;

Case *sequencing* = `overlapping`**:** return the start of the P-partition containing t;

Case *sequencing* = `with_gaps`**:** if t is the start of the P-partition containing t then return t, otherwise return the start of the following P-partition.

Case *asGranule* = `true`**:**

Case *sequencing* = `sequential`**:** if t is between two granules then return the start of the next granule, otherwise return t.

Case *sequencing* = `overlapping`**:** if t is between two granules then return the start of the next granule, otherwise return the start of the granule containing t.

Case *sequencing* = `with_gaps`**:** if t is between two granules then return the start of the next granule. If t is the start of a granule then return t, otherwise return the start of the granule which follows the granule containing t.

The corresponding function $retract(t, P, sequencing, asGranule)$ computes for the start time t of a (backward) split part the end time of the next split part:

Case *asGranule* = `false`**:**

Case *sequencing* = `sequential`**:** return t;

Case *sequencing* = `overlapping`**:** return the end of the P-partition containing t;

Case *sequencing* = `with_gaps`**:** if t is the end of the P-partition containing t then return t, otherwise return the end of the previous P-partition.

Case *asGranule* = `true`**:**

Case *sequencing* = `sequential`**:** if t is between two granules g_1 and g_2 then return the end of g_1, otherwise return t.

Case *sequencing* = `overlapping`**:** if t is between two granules g_1 and g_2 then return the end of g_1, otherwise return the end of the granule containing t.

Case *sequencing* $=$ `with_gaps`**:** if t is between two granules g_1 and g_2 then return the end of g_1. If t is the end of a granule then return t, otherwise return the end of the granule which is before the granule containing t.

■

Back to the function
`split`$(I, duration, asGranule, dateOriented, initialObject, operation,$
$combination, region, forward, inclusion, sequencing, intersect, endTest)$.

The parameters I, *initialObject*, *operation* and *combination* have the same meaning as for the `componentwise` function (Def. 3.73).

The interval I can be split in forward direction ($forward =$ `true`) or in backward direction ($forward =$ `false`).

Region to be split:
The parameter *region* (= `core`, `support` or `kernel`) determines the region $[a, b[$ of the interval I which is to be split. a is the leftmost point of the region and b is the rightmost point of the region. An error is thrown if a or b is the infinity.

The split loop:
Let P_0 be the first partitioning which occurs in *duration*. Let $A = initialObject$ be the accumulator for the operation.

Case $forward =$ `true`**:**

Let $t_0 = startpoint(a, P_0, inclusion, asGranule)$ be the starting point for the split. The `split` command performs the following (forward) loop:

```
$while(t_0 < b)\{$
    let $t_1 := shift(t_0, duration, asGranule, dateOriented)$;   (Def. 3.28)
    let $J := [t_0, t_1[$ be the split part;
    $if\ intersect =$ true $J := I \cap J$;
    $A := union(A, operation(J))$;
    $if(endTest(A) =$ true$)\ return\ A$;
    $t_0 = advance(t_1, P, sequencing, asGranule);\}$
$return\ A$
```

Case $forward =$ `false`**:**

Let $t_1 = endpoint(b, P_0, inclusion, asGranule)$ be the starting point for the split.

The `split` command now performs the following (backwards) loop:

```
$while(t_1 > a)\{$
    let $t_0 := shift(t_1, neg(duration), asGranule, dateOriented)$;   (Def. 3.28)
    let $J := [t_0, t_1[$ be the split part;
    $if\ intersect =$ true $J := I \cap J$;
    $A := union(A, operation(J))$;
    $if(endTest(A) =$ true$)\ return\ A$;
    $t_0 = retract(t_1, P_0, sequencing, asGranule);\}$
$return\ A$
```

4 Summary

The GeTS language is a special purpose functional specification and programming language for temporal notions. It has a basic set of general purpose functional and imperative programming language features. In addition there are a number of built-in data structures and functions which are specific for this application. The most important ones are time points, fuzzy temporal intervals and labelled partitionings of the time line.

GeTS is not a stand alone programming language. It must be part of a host system which provides these data structures and which invokes the GeTS application programming interface.

The GeTS constructs were carefully chosen as a compromise between simplicity and easy usage. Future applications will show whether this goal has been achieved.

References

[1] T. Berners-Lee, M. Fischetti, and M. Dertouzos. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web.* Harper, San Francisco, September 1999. ISBN: 0062515861.

[2] Claudio Bettini, Sushil Jajodia, and Sean X. Wang. *Time Granularities in Databases, Data Mining and Temporal Reasoning.* Springer Verlag, 2000.

[3] François Bry, Bernhard Lorenz, Hans Jürgen Ohlbach, and Stephanie Spranger. On reasoning on time and location on the web. In N. Henze F. Bry and J. Malusyński, editors, *Principles and Practice of Semantic Web Reasoning*, volume 2901 of *LNCS*, pages 69–83. Springer Verlag, 2003.

[4] Nachum Dershowitz and Edward M. Reingold. *Calendrical Calculations.* Cambridge University Press, 1997.

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

[6] Hans Jürgen Ohlbach. About real time, calendar systems and temporal notions. In H. Barringer and D. Gabbay, editors, *Advances in Temporal Logic*, pages 319–338. Kluwer Academic Publishers, 2000.

[7] Hans Jürgen Ohlbach. Calendar logic. In I. Hodkinson D.M. Gabbay and M. Reynolds, editors, *Temporal Logic: Mathematical Foundations and Computational Aspec ts*, pages 489–586. Oxford University Press, 2000.

[8] Hans Jürgen Ohlbach. Geotemporal reasoning: Basic theory. Deliverable D1 of EU NoE Rewerse Working Group A1, 2004.

[9] Hans Jürgen Ohlbach. Relations between fuzzy time intervals. In C. Combi and G. Ligozat, editors, *Proc. of the 11th International Symposium on Temporal Representation and Reasoning*, pages 44–51, Los Alamitos, California, 2004. IEEE.

[10] Hans Jürgen Ohlbach. The role of labelled partitionings for modelling periodic temporal notions. httpd://www.informatik.uni-muenchen.de/mitarbeiter/ohlbach/homepage/publications/PRP/abstracts.shtml, 2004. To be published.

[11] Hans Jürgen Ohlbach. The role of labelled partitionings for modelling periodic temporal notions. In C. Combi and G. Ligozat, editors, *Proc. of TIME 2004*, pages 60–63, Los Alamitos, California, 2004. IEEE.

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

Appendix

A Overview over the Language Constructs

The language constructs are summarized and briefly explained.

A.1 Types

Data Structure Types

`Integer`	standard integers
`Time`	very long integers
`Float`	standard floating point numbers
`String`	strings
`Interval`	fuzzy intervals
`Partitioning`	partitionings
`Label`	labels for partitions
`Duration`	durations
`DateFormat`	date formats

Enumeration Types

type name	possible values
`Bool`	`true/false`
`Side`	`left/right`
`PosNeg`	`positive/negative`
`UpDown`	`up/down`
`IntvRegion`	`core/kernel/support`
`PointRegion`	`core/kernel/support/maximum`
`Hull`	`core/kernel/support/crisp/monotone/convex`
`Fuzzify`	`linear/gaussian`
`Inclusion`	`subset/overlaps/bigger_part_inside`
`SplitInclusion`	`align/subset/overlaps/bigger_part_inside`
`Sequencing`	`sequential/overlapping/with_gaps`
`SDVersion`	`Kleene/Lukasiewicz/Goedel`

A.2 Arithmetics

Binary Arithmetic Operators
The operators are + (addition), `-` (subtraction), `*` (multiplication), (division), % (modulo), `max`, `min`, `pow` (exponentiation) (Def. 3.14).

Unary Arithmetic Operators
`-` (negation), `float`(b) (`Bool` $\mapsto$ `Float`), `round`(a), `round`(a,`up/down`) (Def. 3.15).

Comparisons
<, <=, >, >= (Def. 3.16).
==, != (Def. 3.17).

A.3 Boolean Operators

`-` (complement), `and` or '&&' (conjunction), `or` or '||' (disjunction), `xor` or '^' (exclusive or) (Def. 3.18).

A.4 Control Constructs

`if` c `then` a `else` b (Def. 3.19).

`case` $C_1 : E_1, ..., C_n : E_n$ `else` D (Def. 3.20).

`while` c $\{E_1, ..., E_n\}$ D (Def. 3.21).

`Let` $variable = expression1$ `in` $expression2$ (local binding) (Def. 3.22).

`dLet` $year|month|... =$ `date`$(time, dateFormat)$ `in` $expression$ (local binding of dates) (Def. 3.65).

$x := E$ (assignment) (Def. 3.23).

A.5 Time Points

`now()` of type `Time` (current moment in time) (Def. 3.25)

`shift`$(time, duration, asGranule, dateOriented)$ of type `Time` $*$ `Duration` $*$ `Bool` $*$ `Bool` $\mapsto$ `Time` (time shift by a duration) (Def. 3.28)

`shiftLength`$(time, duration, asGranule, dateOriented)$ of type `Time` $*$ `Duration` $*$ `Bool` $*$ `Bool` $\mapsto$ `Time` (length of a time shift by a duration) (Def. 3.28)

`isInfinity(time)`	`Time` $\mapsto$ `Bool`
`isInfinity(time,positive/negative)`	`Time` $*$ `PosNeg` $\mapsto$ `Bool` (Def. 3.36)

`length`$(t_1, t_2, partitioning, asGranule)$ of type `Time` $*$ `Time` $*$ `Partitioning` $*$ `Bool` $\mapsto$ `Float` (length between t_1 ,t_2 in terms of the partitioning or granule) (Def. 3.42)

A.6 Intervals

[] of type `Interval` (empty interval)

[t1,t2] of type `Interval` (new crisp interval from t1 until t2)

`pushback`$(I, time, value)$ of type `Interval` $*$ `Time` $*$ `Float` $\mapsto$ `Void` adds $(time, value)$ to the membership function of the interval (Def. 3.30).

Set Operations on Intervals

`complement`(I)
  of type `Interval` $\mapsto$ `Interval`
`complement`(I, λ)
  of type `Interval` $*$ `Float` $\mapsto$ `Interval`
`complement`$(I, negation\_function)$
  of type `Interval` $*$ (`Float` $\mapsto$ `Float`) $\mapsto$ `Interval` (Def. 3.31)

`union`(I, J)
of type `Interval` $*$ `Interval` $\mapsto$ `Interval`
`union`(I, J, β)
of type `Interval` $*$ `Interval` $*$ `Float` $\mapsto$ `Interval`
`union`(I, J, co_norm)
of type $Interval$ $*$ `Interval` $*$ (`Float` $*$ `Float` $\mapsto$ `Float`) $\mapsto$ `Interval` (Def. 3.32)

`intersection`(I, J)
of type `Interval` $*$ `Interval` $\mapsto$ `Interval`
`intersection`(I, J, γ)
of type `Interval` $*$ `Interval` $*$ `Float` $\mapsto$ `Interval`
`intersection`$(I, J, norm))$
of type `Interval` $*$ `Interval` $*$ (`Float` $*$ `Float` $\mapsto$ `Float`) $\mapsto$ `Interval` (Def. 3.33)

`setdifference`(I, J)
of type `Interval` $*$ `Interval` $\mapsto$ `Interval`
`setdifference`$(I, J, version)$
of type `Interval` $*$ `Interval` $*$ `SDVersion` $\mapsto$ `Interval`
`setdifference`$(I, J, intersection, complement)$
of type `Interval` $*$ `Interval` $*$ (`Interval` $*$ `Interval` $\mapsto$ `Interval`)$*$
(`Interval` $\mapsto$ `Interval`) $\mapsto$ `Interval` (Def. 3.34)

Predicates on Intervals

`isCrisp`(I)	`Interval` $\mapsto$ `Bool`
`isCrisp`$(I,$ `left/right`$)$	`Interval` $*$ `Side` $\mapsto$ `Bool`
`isEmpty(I)`	`Interval` $\mapsto$ `Bool`
`isConvex(I)`	`Interval` $\mapsto$ `Bool`
`isMonotone(I)`	`Interval` $\mapsto$ `Bool`
`isInfinite(I)`	`Interval` $\mapsto$ `Bool`
`isInfinite(I,left/right)`	`Interval` $*$ `Side` $\mapsto$ `Bool` (Def. 3.35)

`during`$(time, I,$ `core/kernel/support`$)$ of type `Time` $*$ `Interval` $*$ `IntvRegion` $\mapsto$ `Bool` checks whether $time$ is in the corresponding region of the interval I (Def. 3.37).

`isSubset`$(I, J,$ `core/kernel/support`$)$ of type `Interval` $*$ `Interval` $*$ `IntvRegion` $\mapsto$ `Bool` checks whether the corresponding region of I is a subset of the corresponding region of J (Def. 3.37).

`doesOverlap`$(I, J,$ `core/kernel/support`$)$ of type `Interval` $*$ `Interval` $*$ `IntvRegion` $\mapsto$ `Bool` checks whether the corresponding region of I overlaps with the corresponding region of J (Def. 3.37).

`member`$(time, I)$ of type `Time` $*$ `Interval` $\mapsto$ `Float` (membership function) (Def. 3.38).

`components`(I) of type `Interval` $\mapsto$ `Integer` (number of components of I) (Def. 2.2).

`component`(I, k) of type `Interval` $*$ `Integer` $\mapsto$ `Interval` (kth component of I) (Def. 2.2).

`size`(I) of type `Interval` $\mapsto$ `Time` (size of the interval) (Def. 3.40)

`size`$(I, region)$ of type `Interval`$*$`IntvRegion` $\mapsto$ `Time` (size of the corresponding region of the interval) (Def. 3.40)

`size`(I, t_1, t_2) of type `Interval` $*$ `Time` $*$ `Time` $\mapsto$ `Time` (size of the interval between t_1 and t_2) (Def. 3.40)

`sup`(I) of type `Interval` $\mapsto$ `Float` (supremum of I) (Def. 3.41) `inf`(I) of type `Interval` $\mapsto$ `Float` (infimum of I) (Def. 3.41)

`point`$(I, side, region)$ of type `Interval` $*$ `Side` $*$ `PointRegion` $\mapsto$ `Time` (position of the corresponding end of the region) (Def. 3.44).

`centerPoint`(I, n, m) of type `Interval` $*$ `Integer` $*$ `Integer` $\mapsto$ `Time` (n-m center point) (Def. 3.46).

Manipulation of Intervals

`shift`(I, t) of type `Interval` $*$ `Time` $\mapsto$ `Interval` shifts the interval by the given time (Def. 3.47).

`cut`(I, t_1, t_2) of type `Interval` $*$ `Time` $*$ `Time` $\mapsto$ `Interval` (extracts the part of I between t_1 and t_2) (Def. 3.48).

`hull`$(I,$ `core/support/kernel/crisp/monotone/convex`$)$ of type `Interval`$*$`Hull` $\mapsto$ `Interval` (construction of the corresponding hull) (Def. 3.49).

`invert`(I) of type `Interval` $\mapsto$ `Interval` inverts the membership function (Def. 3.50).

`scaleup`(I) of type `Interval` $\mapsto$ `Interval` scales the membership function up to maximal value 1 (Def. 3.51).

`times`(I, f) of type `Interval` $*$ `Float` $\mapsto$ `Interval` multiplies the membership function of I with f (Def. 3.52).

`exp`(I, e) of type `Interval` $*$ `Float` $\mapsto$ `Interval` exponentiates the membership function of I with e (Def. 3.52).

`extend`$(I,$ `positive/negative`$)$ of type `Interval`$*$`PosNeg` $\mapsto$ `Interval` extends I to the infinity (Def. 3.53).

`extend`$(I, length, side)$ of type `Interval` $*$ `Time` $*$ `Side` $\mapsto$ `Interval` extends or shrinks I (Def. 3.54).

`integrate`$(I,$ `positive/negative`$)$ of type `Interval`$*$`PosNeg` $\mapsto$ `Interval` integrates the membership function (Def. 3.55).

`fuzzify`$(I,$ `linear/gaussian`, `left/right`, $increase$, $offset)$
of type `Interval`, `Fuzzify`, `Side`, `Float`, `Float` $\mapsto$ `Interval`
`fuzzify`$(I,$ `linear/gaussian`, `left/right`, $x1, x2, offset)$
of type `Interval`, `Fuzzify`, `Side`, `Time`, `Time`, `Time` $\mapsto$ `Interval` (Def. 3.57)
fuzzifies the interval at the given side with the given fuzzification function.

`integrateSymmetric`$(I, J, simple)$
of type`Interval` $*$ `Interval` $*$ `Bool` $\mapsto$ `Float` and
`integrateAsymmetric`(I, J)
of type `Interval` $*$ `Interval` $\mapsto$ `Float`
symmetric or asymmetric integration of the membership function of I over the membership function of J (Def. 3.59).

`MaximizeOverlap`(I, J, EJ, D) of type `Interval`$*$`Interval`$*$`Interval`$*$(`Interval`$*$`Interval` $\mapsto$ `Float`) $\mapsto$ `Float` (Def. 3.61)

A.7 Time and Partitions

`time`$(year|month|..., dateFormat)$ of type `Integer`$|\ldots|$`Integer` $*$ `DateFormat` $\mapsto$ `Time` maps a date in a given date format to the time point denoted by this date (Def. 3.64).

`partition`(*time*, *partitioning*) of type `Time` $*$ `Partitioning` $\mapsto$ `Interval`
`partition`(*time*, *partitioning*, n, m) of type
`Time` $*$ `Partitioning` $*$ `Integer` $*$ `Integer` $\mapsto$ `Interval`
compute partitions as intervals (Def. 3.66).

`partitionBoundary`(*time*, *partitioning*, `left/right`) of type `Time` $*$ `Partitioning` $*$ `Side` $\mapsto$ `Time` compute partition boundaries (Def. 3.67).

`partitioningIsBounded`(P, `left/right`) of type `Partitioning`$*$`Side` $\mapsto$ `Bool` checks whether the valid region of the partitioning is bounded (Def. 3.68).

`partitioningBoundary`(P, `left/right`) of type `Partitioning` $*$ `Side` $\mapsto$ `Time` returns the boundaries of the valid region of the partitioning (Def. 3.68).

`which`(*time*, P, Q, *inclusion*, *asGranule*) of type `Time`$*$`Partitioning`$*$`Partitioning`$*$`Inclusion`$*$ `Bool` $\mapsto$ `Time` (which week in the year, for example) (Def. 3.69)

Labels
`label`(*time*, *partitioning*) of type `Time` $*$ `Partitioning` $\mapsto$ `Label` returns the label of the corresponding partition (Def. 3.70).

`isLabel`(*label*) of type `Label` $\mapsto$ `Bool` checks whether the label is not the NULL label (Def. 3.70).

`isGap`(*label*) of type `Label` $\mapsto$ `Bool` checks whether the label is the gap label (Def. 3.70).

`LabelName`(*string*) of type `String` $\mapsto$ `Bool` turns the string into a label (Def. 3.70).

`extractLabelled`(I, *label*, *partitioning*, *inclusion*, *intersect*) of type `Interval`$*$`Label`$*$`Partitioning`$*$ `SplitInclusion` $*$ `Bool` $\mapsto$ `Interval` extracts partitions in the interval I with the given label (Def. 3.71).

`nextGranule`(*time*, *partitioning*, *label*, n, *withGaps*) of type `Time` $*$ `Partitioning` $*$ `Label` $*$ `Integer` $*$ `Bool` $\mapsto$ `Interval` constructs a new interval which represents a granule (Def. 3.72).

Loops over Intervals
`componentwise`(I, *initialObject*, *operation*, *combination*) of type `Interval` $*$ T $*$ (`Interval` $\mapsto$ T) $*$ ($T * T \mapsto T$) $\mapsto T$ applies *operation* to all components of the interval (Def. 3.73).

`split`(I, *duration*, *asGranule*, *dateOriented*, *initialObject*, *operation*,
combination, *region*, *forward*, *inclusion*, *sequencing*, *intersect*) of type
`Interval` $*$ `Duration` $*$ `Bool` $*$ `Bool` $*$ T $*$ (`Interval` $\mapsto$ T) $*$ (`Interval` $*$ `Interval` $\mapsto$ T) $*$ `IntvRegion` $*$ `Bool` $*$ `SplitInclusion` $*$ `Sequencing` $*$ `Bool` $\mapsto T$
splits the interval into parts and applies the operation to them (Def. 3.74).

B The Application Programming Interface

The C++ API of the GeTS language is as follows:

GeTS functions are realized as a class `Function` in a namespace `GeTS`. They can be defined, they can be applied to arguments, and some information about them can be retrieved.

Definition:
A new GeTS function can be created with an ordinary constructor:
`fct = new Function(definition).`

The `definition` is a string representation of the definition, optionally followed by the keyword `explanation` and some text. The explanation can be retrieved just by `fct->explanation`.

The definition is parsed and compiled. Parsing or compilation errors can be obtained by `fct->getError()`. The function `fct->noError()` checks whether there was a parsing or compilation error.

Information about Functions:
The function definitions can be obtained in different versions:

`fct->callString()` returns the function call as string

`fct->typeString()` returns the function type as string

`fct->definitionString()` returns the function definition with line numbering as string.

`fct->codeString()` returns the abstract machine code as string.

Example B.1 (for `codeString()`) The code string for the function

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

(Example 3.3) is

```
0: I[1,Interval]
1: isEmpty(Interval->Bool)
2: ||(Bool*Bool->Bool)
3: I[1,Interval]
4: left[-1,left,Side]
5: isInfinite(Interval*Side->Bool)
6: ||(Bool*Bool->Bool)
7: IfThenElse(Bool*Bool*Bool->Bool)
8: false[-1,false,Bool]
9: IfThenElse(Bool*Bool*Bool->Bool)
10: t[0,Time]
11: I[1,Interval]
12: left[-1,left,Side]
13: support[-1,support,IntvRegion]
14: point(Interval*Side*IntvRegion->Time)
15: <(Time*Time->Bool)
16: IfThenElse(Bool*Bool*Bool->Bool)
```

It should be fairly obvious what this means. For example, line 0, `I[1,Interval]` means that the parameter `I` at parameter position 1 and of type `Interval` is pushed to the stack. Line 1: `isEmpty(Interval->Bool)` means that the `isEmpty` predicate pops its argument from the stack, performs the check, and pushes the result to the stack again. Line 2: `||(Bool*Bool->Bool)` is the first invocation of the `or` check. It checks the top of stack. If this is the Boolean value `true` then this value is popped from the stack and the program counter is set to 7. The remaining program steps are more or less self explaining. ■

It should be noticed that the actual computations, for example, integrating over a membership function of an interval, are done with compiled machine code. The commands of the GeTS abstract machine are only used to control the invocation of this machine code.

Auxiliary Classes and Types:

The data types of GeTS are represented as a class `Type` in the namespace `GeTS`. They can be basic data types or compound types. The most important API method for types is `toString`. Most other methods are for internal use.

The data which are manipulated by a GeTS function are comprised into a union type. Without further explanation we just list the definition.

```
union GeTSValue {
  long long int*        Time;
  PartLib::Partitioning* Partitioning;
  PartLib::Label*       Label;
  PartLib::DateFormat*  DateFormat;
  FuTIRe::Interval*     Interval;
  Function*             lFunction;
  int                   Integer;
  float                 Float;
  bool                  Bool;
  DurationSpec*         Duration;
  string*               String;
};
```

Application:
There are two application functions:
`pair<Type*, GeTSValue> apply(vector<pair<Type*, GeTSValue> >& values)`
can be used to apply the function to a vector of parameters. The result is a pair consisting of the result type and the result value.

The other method

```
pair<Type*, GeTSValue>
   apply(const string& arguments,const vector<FuTIRe::Interval*>& intervals)
```

can be used to apply the function to a string representation of the parameters. Intervals are represented as non-negative integers. The integers are used as indices in the given vector of interval pointers. The result is again a type-value pair.