GeTS – A Specification Language for Geo-Temporal Notions

Hans Jürgen Ohlbach

Institut für Informatik, Universität München
E-mail: ohlbach@lmu.de

Abstract. This paper contains a brief overview of 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 labeled partitionings of the time axis. The partitionings are used to represent periodic temporal notions like months, semesters etc. and also whole calendar systems. GeTS is essentially a typed functional language with a few imperative constructs and many built-ins. 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. A parser, a compiler and an abstract machine for GeTS is implemented.

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' [7], 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 [5], mainly for dealing with temporal information in databases. This work is becoming more important now with the emergence of the Semantic Web [2]. 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–project ('Computational Treatment of Temporal Notions') [16] we aim at a very

detailed modeling 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 data types for elementary temporal notions. These are time points, crisp and fuzzy time intervals [20] and partitionings for representing periodical temporal notions like years, months, semesters etc. [22]. The partitionings can be specified algorithmically or algebraically. The algorithmic specifications allow 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'[1]) for specifying complex temporal notions. A very first version of this language has been presented in [17, 18], but the new version has more than 20 times as many constructs and features than the old one. 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–system. GeTS is the first specification and programming language with such a rich variety of built-in data structures and functions for geo-temporal notions. The details of the language can be found in the technical report [21]. The third layer contains interfaces to GeTS and the other modules of the system. The standard interface is socket based. There are experimental RMI, SOAP and CORBA interfaces [1].

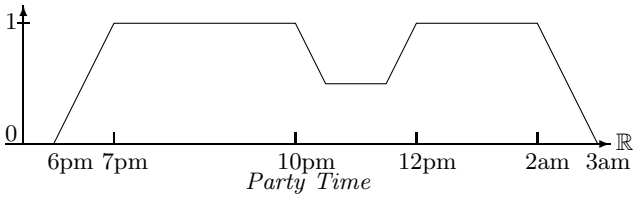
2 Basic Data Structures in CTTN and GeTS

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.

[1] The prefix 'geo' in the word geo-temporal was chosen to distinguish it from temporal logics in the usual understanding. 'geo, i.e. 'earth', emphasizes that it is about temporal notions as used on our planet. There is a close analogy to the area of 'geo-spatial' in contrast to 'spatial' representation and reasoning.

The next important datatype is that of time intervals. Time intervals can be crisp or fuzzy [27, 8]. 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.



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 CTTN–system has an extensive implementation of fuzzy time intervals with a rich application programming interface [20].

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 modeled as partitionings.

Partitionings of the time axis are infinite mathematical structures. Therefore they must be represented on a computer in a more indirect way. We distinguish three aspects of partitionings of the time axis:

1. the mathematical structure. It serves as the semantics for the more concrete descriptions of these objects;
2. the specification of concrete partitionings. There are different ways to specify them. Each type of specification comes with a mathematical structure that has also a serialized text form which can be stored in files;
3. the implementation. There is a common interface for all types of partitionings, such that the algorithms working with these partitionings are independent of the specification type. The methods of the partitioning application interface (API) are automatically compilable from the specification.

Different types of specifications for partitionings and a common API for all of them is implemented in the PartLib library [22]. The first type of partitionings are called 'algorithmic partitionings'. They are characterized by implementing an isomorphism to integers directly. All the standard periodic temporal notions, years, months, weeks etc., but also Easter time, sun rises, tides etc. are of this

type. The implementation can in particular take into account all the nasty and irregular features of real calendar systems, leap years, leap seconds, daylight savings time, time zones etc.

Another type of specification are called 'duration partitionings'. They are specified by giving an anchor date and a list of durations. For example, one can specify semesters in this way. The anchor date could be first of October 2000. The durations could be '6 months' (for the winter semester) and '6 months' (for the summer semester). A further type are 'date partitionings', which are specified by concrete dates. An example could be the seasons. 2000/3/21 spring 2000/6/21 summer 2000/9/23 autumn 2000/12/21 winter 2001/3/21 specifies the seasons for one year. An extrapolation mechanism extrapolates them to the infinity. In principle, all partitionings are infinite, but there is a mechanism for constraining a 'validity region'. This way one can express, for example, 'I have this meeting every Monday 9:00-10:00 for the next 15 weeks'.

The next version of the PartLib library will contain 'tree partitionings' [23]. A bus timetable, for example, can be specified as a tree partitioning: '(in very winter, in every week, (in day 0-4, hour 5, minute 20, bus B1, hour 6, minute 20 bus B2 ...), (in day 5-6, hour 8, minute 20 bus B1, ...)), (in every spring ...)...

Partitions can be *labeled*. 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 modeled by taking an 'hour' partitioning, and attached labels 'working_hour' and 'gap' to the hour partitions. Groups of hour partitions labeled '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'. A group of partitions with the same label, possibly interrupted by 'gap'–partitions, is a *granule*.

Remark 1 (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. ■

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. $d = -2\ week + 3\ month$, for example, denotes a backward shift of 2 weeks followed by a forward shift of 3 months.

3 The GeTS Language

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

- 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.
- 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. This excludes using an existing functional language like SML or Haskell.
- 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.
- Developing GeTS from scratch has also the advantage that one can design the syntax of the language in a way which better reflects the semantics of the language constructs. As an example, the syntax for a time interval constructor is just $[expression_1, expression_2]$.

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

Example 1 (tomorrow). The definition

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

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

Example 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 interval between t_1 and t_2. The result is therefore the 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 (Point–Interval Before Relation). The function

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

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

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

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

Example 4 (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 the relation IIRFuzzyBefore. 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. The 'effect' function turned out a useful test and debug tool for developing the fuzzy interval–interval relations [19, 24].

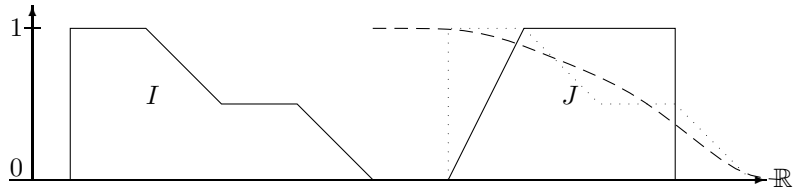
Example 5 (effect).

```
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 $T_1 * \ldots * T_n \mapsto T$.

Definition 1 (Data Structure Types).

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

■

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.

`Interval`s are realized as *polygons* with integer coordinates. An interval is therefore a sequence of pairs $I = (x_0, y_0), \ldots, (x_n, y_n)$. The x_i are `Time` points and the y_i are fuzzy values. Internally the y_i are realized as short integers between 0 and 1000. From the GeTS point of view, however, the y_i are `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 [20].

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

`Duration`s 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 are sequences $P_0/\ldots/P_n$ of Partitionings.

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.

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.

The language has an operational semantics. It is described in detail in [21] where all language constructs are introduced. 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 2 (Function Definitions). *A GeTS function definition has one of the forms*

(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) $type : name(type_1\ var_1, \ldots, type_n\ var_n)$

■

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

Function definitions can be overloaded. They are distinguished by their argument types, not by the result type.

Standard Language Constructs GeTS supports the same kind of arithmetic and Boolean expressions as many other programming languages. A small difference is the `Time` type, which is integrated in the arithmetics of GeTS. The obligatory 'if-then-else' construct is of course also available. 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. The values of local variables can be changed with an assignment construct. The assignment operation returns no value. It can only occur in the body of the `while` statement.

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.

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 $type_1 * \ldots * type_n$, (for distinguishing between overloaded functions), or
3. a lambda expression:
   $lambda(type_1\ variable_1, \ldots, type_n\ variable_n)\ expression.$ '*expression*' can contain variables which are lexically bound outside the parameter list of *lambda*.

3.3 Built-ins for 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. The expression $[t_1, t_2]$ of type `Time*Time` $\mapsto$ `Interval`, for example,

constructs a new crisp interval with boundaries t_1 and t_2. `pushBack`$(I, time, value)$ of type `Interval` $*$ `Time` $*$ `Float` $\mapsto$ `Void` extends a fuzzy interval with a new point for the envelope polygon.

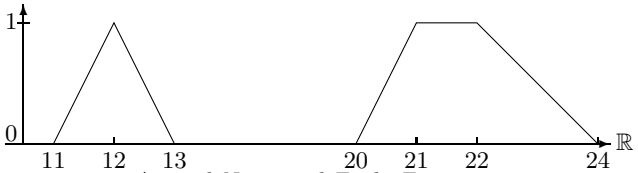
For crisp intervals there are the standard set operators: complement, intersection, union etc. These are uniquely defined. There is no choice. Unfortunately, or fortunately, because it gives you more flexibility, there are no such uniquely defined set operators for fuzzy intervals. Set operators are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

Predicates like 'isCrisp', 'isEmpty', 'isConvex', 'isMonotone', 'isInfinite' can be used to check the corresponding properties of intervals. Basic relations between time points and intervals, or between key parts of two intervals can be checked with predicates like 'during' or 'subset' etc. The function `member`$(time, I)$ returns the fuzzy membership value for an interval. If an interval is non-convex, there a number of functions to count components, measure their size, extract them or map over them. n, m*-center points* are used to express temporal notions like 'the first half of the year', or 'the second quarter of the weekend'. This can be computed with the function `centerPoint`(I, n, m).

Intervals can be transformed in various ways: shifted, scaled, extended or shrinked, integrated or fuzzified with linear or Gaussian shapes. Parts can be cut out, it can be split into different parts. Different types of hulls can be calculated. Membership functions can be multiplied or exponentiated.

Example 6 (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:

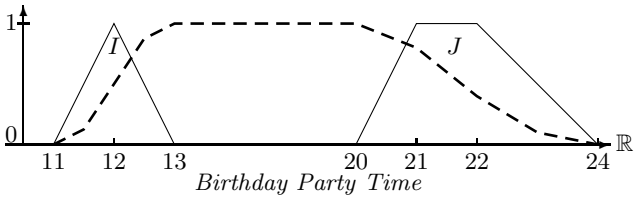


Around Noon and Early Evening

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 t is 1 if the party definitely started before t and definitely ended after t. Therefore the fuzzy value at point t is computed by integrating over the membership functions of the start intervals and the end intervals. A natural definition would therefore be:

```
partyTime(Interval I, Interval J)
= intersection(integrate(I,positive),integrate(J,negative))
```
(1)

The resulting fuzzy set is:



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

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` = 10 means that the duration of the actual linear increase should be 10% 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 (which is at 10% of the distance between 2004/9/28 and Christmas) and then stays at 1.0 until 2004/12/25.

Example 7 (Before Christmas).

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

■

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 and later).

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. A time point or an interval can be shifted by a fraction of a partitioning (1.5 years, for example). Two different shift functions are available, a length oriented shift and a date oriented shift, which give slightly different results for fractional shifts.

Date and Time In examples 2 and 7 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. Examples are:
`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 first week in 2004 started at Monday, 29th of December 2003).

The `dLet` construct is a kind of inverse to the `time` function. The expression `dLet` $year|month|... =$ `date`$(time, dateFormat)$ `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. **Example:** '`dLet` $y|m|d|h =$ `date`$(0,$ `Gregorian_month`$)$ `in` $y + m + d$' yields 1973 because the time point 0 corresponds to the first of January 1970. Therefore $y = 1970$, $m = 1$, $d = 1$ and $h = 0$.

Partitionings and Labels. GeTS has a number of functions for reckoning with time points, partitions and labels. The function `partition`$(time, partitioning)$ maps time points to intervals, which represent partitions.

The version `partition`$(time, partitioning, n, m)$ 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.

The next function is `which`$(time, P, Q, inclusion, asGranule)$. It can, for example, be used to compute which week in the year is now, or which day in the semester is now.

The further set of functions deals with labels of partitions. Labels are not just strings, but also special data structures. `label`$(time, partitioning)$ returns the label of the partition containing $time$. If there is no labeling defined, it returns a NULL label.

The function `isLabel`(*label*) checks whether the *label* is not the NULL label. `isGap`(*label*) checks whether the label is the gap label. `LabelName`(*name*) turns a string (without quotes) into a `Label`.
The function `extractLabelled`($I, label, partitioning, inclusion, intersect$) can be used to extract from an interval all partitions with a given label, for example all Tuesdays of a labeled day partitioning. The `extractLabelled` function maps through all partitions of the given partitioning which are labeled 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* = `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 `nextGranule`($time, partitioning, label, n, withGaps$) is for constructing intervals which represent granules. The interval is constructed as follows: If *time* is inside a granule with the given *label* and if $n = 0$ then this granule is computed. Otherwise the n^{th} next/previous (if $n < 0$) granule with this label is computed. If *time* lies outside a granule with the given *label* and $n = 0$ then the empty interval is returned. Otherwise the n^{th} next/previous (if $n < 0$) granule with this label is computed.

4 Summary and Related Work

Most of the approaches for modeling every-day temporal notions are 'monolithic', i.e. there is one single formalism for specifying calendar systems. In particular there is all the work about the mathematical representation of periodic temporal notions as *time granularities*, or similar kind of mathematical objects. A good overview is given in the book of Bettini, Jajoda and Wang [5]. This work is particularly motivated by the need to represent time in temporal databases. A selection of papers about the abundant work in this area is [3, 15, 12, 25, 13, 14, 9, 4, 10, 6, 11, 26]. In contrast to these approaches, the CTTN–system has various representation formalisms for the different aspects of temporal notions. One of the components is the GeTS language as 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 labeled 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. In a first application, various versions of fuzzy binary relations between fuzzy intervals have been defined [24]. Example 4 (fuzzy before) is one of them.

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

References

1. Julius Benkert. Integration of the CTTN system in Java. Master's thesis, Inst. for Computer Science, LMU Munich, 2006.
2. T. Berners-Lee, M. Fischetti, and M. Dertouzos. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web.* Harper, San Francisco, September 1999. ISBN: 0062515861.
3. C. Bettini and R.D.Sibi. Symbolic representation of user-defined time granularities. *Annals of Mathematics and Artificial Intelligence*, 30:53–92, 2000. Kluwer Academic Publishers.
4. Claudio Bettini, Curtis E. Dyreson, William S. Evans, Richard T. Snodgrass, and X. Sean Wang. *Temporal Databases, Rreseach and Practice*, volume 1399 of *LNCS*, chapter A Glossary of Time Granularity Concepts, pages 406–413. Springer Verlag, 1998.
5. Claudio Bettini, Sushil Jajodia, and Sean X. Wang. *Time Granularities in Databases, Data Mining and Temporal Reasoning.* Springer Verlag, 2000.
6. Claudio Bettini, Sergio Mascetti, and X. Sean Wang. Mapping calendar expressions into periodical granularities. In C. Combi and G. Ligozat, editors, *Proc. of the 11th International Symposium on Temporal Representation and Reasoning*, pages 87–95, Los Alamitos, California, 2004. IEEE.
7. Nachum Dershowitz and Edward M. Reingold. *Calendrical Calculations.* Cambridge University Press, 1997.
8. Didier Dubois and Henri Prade, editors. *Fundamentals of Fuzzy Sets.* Kluwer Academic Publisher, 2000.
9. Curtis E. Dyreson, Wikkima S. Evans, Hing Lin, and Richard T. Snodgrass. Efficiently supporting temporal granularities. *IEEE Transactions on Knowledge and Data Engineering*, 12(4):568–587, 2000.
10. Lavinia Egidi and Paolo Terenziani. A lattice of classes of user-defined symbolic periodicities. In C. Combi and G. Ligozat, editors, *Proc. of the 11th International Symposium on Temporal Representation and Reasoning*, pages 13–20, Los Alamitos, California, 2004. IEEE.
11. I.A. Goralwalla, Y. Leontiev, M.T. Ozsu, D. Szafron, and C. Combi. Temporal granularity: Completing the picture. *Journal of Intelligent Information Systems*, 16(1):41–63, 2001.
12. Nick Kline, Jie Li, and Richard Snodgrass. Specifying multiple calendars, calendric systems and field tables and functions in timeadt. Technical Report TR-41, Time Center Report, May 1999.
13. B. Leban, D. Mcdonald, and D.Foster. A representation for collections of temporal intervals. In *Proc. of the American National Conference on Artificial Intelligence (AAAI)*, pages 367–371. Morgan Kaufmann, Los Altos, CA, 1986.

14. M. Niezette and J. Stevenne. An efficient symbolic representation of periodic time. In *Proc. of the first International Conference on Information and Knowledge Management*, volume 752 of *Lecture Notes in Computer Science*, pages 161–169. Springer Verlag, 1993.
15. Peng Ning, X. Sean Wang, and Sushil Jajodia. An algebraic representation of calendars. *Annals of Mathematics and Artificial Intelligenc*, 36(1-2):5–38, September 2002. Kluwer Academic Publishers.
16. Hans Jüergen Ohlbach. Computational treatement of temporal notions – the CTTN system. In François Fages, editor, *Proceedings of PPSWR 2005*, Lecture Notes in Computer Science, pages 137–150, 2005. see also URL: http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-30.
17. 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.
18. 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.
19. Hans Jürgen Ohlbach. Relations between fuzzy time intervals. In *Proceedings of 11th International Symposium on Temporal Representation and Reasoning, Tatihoui, Normandie, France (1st–3rd July 2004)*, pages 44–51. IEEE Computer Society, 2004. See also http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2004-33.
20. Hans Jürgen Ohlbach. Fuzzy time intervals – the FuTI-library. Research Report PMS-FB-2005-26, Inst. für Informatik, LFE PMS, University of Munich, June 2005. URL: http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-26.
21. Hans Jürgen Ohlbach. GeTS – a specification language for geo-temporal notions. Research Report PMS-FB-2005-29, Inst. für Informatik, LFE PMS, University of Munich, June 2005. URL: http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-29.
22. Hans Jürgen Ohlbach. Modelling periodic temporal notions by labelled partitionings – the PartLib library. In S. Artemov, H. Barringer, A. d'Avila Garces, L. C. Lamb, and J. Woods, editors, *Essays in Honour of Dov Gabbay*, volume 2, pages 453–498. College Publications, King's College, London, 2005. ISBN 1-904987-12-5. See also http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-28.
23. Hans Jürgen Ohlbach. Periodic temporal notions as 'tree partitionings', March 2006. Submitted to PPSWR06.
24. Hans Jürgen Ohlbach. Relations between fuzzy time intervals. Research Report PMS-FB-2006-26, Inst. für Informatik, LFE PMS, University of Munich, June 2006. URL: http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2006-26.
25. Michael D. Soo and Richard T. Snodgrass. Mixed calendar query language support for temporal constants. Technical Report TR 92-07, Dept. of Computer Science, Univ. of Arizona, February 1992.
26. Stephanie Spranger. *Calendars as Types – Data Modeling, Constraint Reasoning, and Type Checking with Calendars.* Dissertation/Ph.D. thesis, Institute of Computer Science, LMU, Munich, Munich, 2005. PhD Thesis, Institute for Informatics, University of Munich, 2005.
27. L. A. Zadeh. Fuzzy sets. *Information & Control*, 8:338–353, 1965.