Towards a Multi-Calendar Temporal Type System for (Semantic) Web Query Languages

François Bry and Stephanie Spranger

University of Munich, Munich 80538, Germany,
bry,spranger@pms.ifi.lmu.de,
WWW home page: http://www.pms.ifi.lmu.de

Abstract. Time is omnipresent on the (Semantic) Web. However, formalism like XML, XML Schema, RDF, OWL and (Semantic) Web query languages have, if any, only very limited notions of temporal data types and temporal theories built-in. Recently, the development of Web Services for temporal operations has begun. In this article, we describe a connection, possibly the first one, between such Web Services and Web formalisms: A proposal of a *type system* for temporal and calendric data, called *multi-calendar temporal type system* seamlessly integrated into a host (query) language. The type system's associated type checking methods are beyond the scope of this article. For proof-of-concept purposes, the Web and Semantic Web query language Xcerpt has been chosen.

1 Introduction

Time is omnipresent on the Web or Semantic Web (for short (Semantic) Web). Many Web sites and pages implicitly or explicitly refer to temporal and calendric data. Many advanced (Semantic) Web applications like web-based information and appointment scheduling systems and so-called adaptive Web systems refer to temporal and calendric data, as well. Most existing or foreseen mobile computing applications refer not only to locations but also to time. For example, a mobile application listing pharmacies in the surrounding of a user will preferably only mention those that are currently open, i.e. it refers to rather sophisticated temporal and calendric data. The temporal and calendric data involved are most often rather complex, sometimes involving different calendars (e.g. cultural calendars like the Gregorian and the Islamic and business calendars) with various regulations and lots of irregularities (e.g. leap years), and 'trimmed to fit' individual use. For example, one might think of the Web sites of a university announcing lectures, courses, examinations, consultation hours of professors, etc. within a teaching term, or a personal appointment book containing entries like business conferences or personal work out times during the

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

summer period. How to represent, query, and process such kind of time and calendar data on the (Semantic) Web, i.e. using Web formalisms like XML, XML Schema, RDF, OWL, and (Semantic) Web query languages?

On the one hand, those formalisms developed for the (Semantic) Web have, if any, only very limited notions of temporal datatypes and temporal theories built-in. The W3C standard XML Schema, for example, supports some temporal data types which are restricted to the Gregorian calendar without multi-calendar reasoning, e.g. no "calendar cast primitives". Furthermore, only some operations over temporal data types for temporal computation and/or reasoning in XQuery implementing XMLSchema are supported. On the other hand, several time and calendar theories have been investigated and developed for a long time from different perspectives: (1) In Artificial Intelligence, temporal logics and calculi and temporal constraint reasoning independent of calendars have been investigated, e.g. [1–5]. (2) In temporal databases, algebraic representations for time and calendars have been theoretically investigated, e.g. [6–8]. It is however not clear, neither how these general methods can be applied to both real time dealing with irregularities (e.g. leap seconds[1] and castings between different calendars) and temporal types, nor how static type checking and type inferencing as needed when types are integrated in programming or query languages can be performed. (3) Several algorithms for calendrical calculations [9] are developed and implemented allowing for various calendar-based calculations. Recently, the development of some Web Services for temporal operations have begun [10, 11].

In this article, we describe a (possibly first) connection between such Web Services and Web formalisms: A proposal of a *type system* for temporal and calendric data, called *multi-calendar temporal type system,* seamlessly integrated into the Web and Semantic Web query language Xcerpt [12]. This article is devoted to the language of the type system. Its associated type checking methods are beyond the scope of this article. Static type checking is as useful and desirable with temporal and calendric data types as it is with whatever other data type. It makes error detection at compile time possible, and it improves code generation, as well. Specific aspects of calendar systems make static type checking for such data an interesting challenge. The temporal type system is *open* inasmuch that the type system is neither restricted to a particular calendar nor to specific temporal or calendric data. Instead, a programmer can specify any professional, cultural, religious, etc. time concept his/her application might need. The type system provides a small, but powerful, declarative set of *type constructors, selectors,* and, as they are sometimes called, *mutators.* Note that, for example the temporal data types defined in XML Schema are specified without such type operators.

This article is organized as follows. Section 1 is this introduction. Section 2 sketches the major concepts of the time model used for the multi-calendar temporal type system. Section 3 introduces the multi-calendar temporal type system: the type constructors and selectors and its module structure, and this section exemplifies the type system's expressiveness. Section 4 gives perspectives

[1] International System of Units(SI), http://www.bipm.org/en/si/

for proof-of-concept of the multi-calendar temporal type system. Finally, Section 5 concludes this article.

2 Time Model

The time model used for the multi-calendar temporal type system combines interval-based temporal logics, in particular Allen's interval calculus [2] with the concept of 'hierarchical time lines' to represent time and so-called calendar units like 'hour' or 'day' (often realized as time granularities, e.g. in [6–8]). In the present time model, however, hierarchical time lines are realized as *time partition*s of a *time domain* and *selections* of time partitions. Additional concepts are *durations* and *time intervals* over those time partitions (resp. selections), improving and simplifying the layered time model presented in [13].

The time domain is the set of time points used to interpret time and calendar concepts. A time point is a non-divisible moment of time which is much smaller than the smallest extend of time which can be modeled in the type system.

Definition 1. (Time Domain.) *The* time domain *is a pair* $(\mathcal{T},<_\mathcal{T})$ *where* $\mathcal{T}$ *is an infinite set and* $<_\mathcal{T}$ *is a total order on* $\mathcal{T}$ *such that* $\mathcal{T}$ *is not bounded for* $<_\mathcal{T}$. *If* $(\mathcal{T},<_\mathcal{T})$ *is a time domain, then an element* $t \in \mathcal{T}$ *is called* time point.

To introduce hierarchical time lines, the time domain can be partitioned into countable finite many, finite or infinite parts in the time domain $(\mathcal{T},<_\mathcal{T})$.

Definition 2. (Time Partition of $(\mathcal{T},<_\mathcal{T})$ and Parts). *Let* $(\mathcal{T},<_\mathcal{T})$ *be a time domain. Then* $\mathcal{P} \subset \mathcal{P}(\mathcal{T})$ *(i.e. the powerset of* $\mathcal{T}$ *is a* time partition of $(\mathcal{T},<_\mathcal{T})$, *if*

1. *all sets of* $\mathcal{P}$ *are non-empty,*
2. *if for all* $t_i \in p$ *and for all* $t_j \in q$ *with* $t_i, t_j \in \mathcal{T}$, p *and* q *sets in* $\mathcal{P}$ $t_i <_\mathcal{T} t_j$, *then* $p < q$,
3. *any two distinct sets of* $\mathcal{P}$ *are disjoint, and*
4. *every time point of* $\mathcal{T}$ *belongs exactly to one of the sets of* $\mathcal{P}$, *i.e.* $\mathcal{T}$ *is the union of the sets of* $\mathcal{P}$.

A set p *of a time partition* $\mathcal{P}$ *is called* part.

With Definition 2, the calendar units `hour, day, week,` and `month` of the Gregorian calendar are time partitions, and `2004-05-05` (using the notation of the ISO 8601 standard for dates and time[2]) is a part of `day.` Note that in a concrete implementation, a part of a time partition may be represented by an interval in the time domain.

Infinite sets of gapped subsets of a time partition are called *selections.* The set of all Saturdays is a selection of the time partition `day,` for example.

Time Partitions are related in the sense that the parts of one time partition may be further aggregated by a specific set of parts of another time partition to form one part in an *including* time partition.

[2] http://www.iso.ch/iso/en/prods-services/popstds/datesandtime.html

Definition 3. (Include Relation). *Let $\mathcal{P}$ and $\mathcal{Q}$ be two time partitions. $\mathcal{P}$* includes *$\mathcal{Q}$ ($\mathcal{P} \trianglerighteq \mathcal{Q}$) if for each part $p \in \mathcal{P}$ there exists a set S of parts in $\mathcal{Q}$ such that $p = S$. Then $\mathcal{Q}$ is* included in *$\mathcal{P}$ ($\mathcal{Q} \trianglelefteq \mathcal{P}$).*

With Definition 3, `day` $\trianglelefteq$ `day, day` $\trianglelefteq$ `week, day` $\trianglelefteq$ `month,` but not `week` $\trianglelefteq$ `month` (if the Gregorian calendar is used). The *include* relation introduces a partial order over time partitions.

The *include* relation permits to specify and relate time partitions in terms of included time partitions as further "aggregations" of the time domain to larger parts. The larger parts are specified by durations of parts of the included time partition which need to be anchored in the included time partition. Of course, if the time partition `day` is included in the time partition `week,` for example, then any selection of `day` is included in `week,` as well.

In the multi-calendar temporal type system, the include relation is realized as a typing relation, called *include typing relation* between types.

To measure the amount of time in a time partition (resp. a selection), for example `4 weeks` (in the time partition `week`), durations are defined.

Definition 4. *Let $\mathcal{P}$ be a time partition or a selection of a time partition. Then a* duration *d in $\mathcal{P}$ is a signed integral number of parts of $\mathcal{P}$, i.e. an amount of time with known length but no specific starting or ending point.*

A time interval is a not necessarily connected, possibly infinite, anchored number of parts of a time partition (resp. selection) with the constraints that the parts in a time interval are totally ordered and disjoint.

Definition 5. *Let $\mathcal{P}$ be a time partition or a selection of a time partition. Then a* time interval *I is a finite or infinite collection of pairwise disjoint, totally ordered intervals $[p, q]$, $p, q \in \mathcal{P}$ where $p \leq q$.*

With Definition 5, `[2003-03,2003-08]` (using the notation of the ISO 8601 standard for dates and time[3]) is a time interval of time partition `month`.

3 The multi-calendar temporal Type System

This section introduces and exemplifies the syntactic forms of the type system, i.e. its type constructors and selectors which are similar to base types and structured types in programming languages like integers, strings, and lists with appropriate operations over these types and the type system's module structure.

3.1 Base Types

In general, a base type is a set of simple, unstructured values such as numbers or booleans with appropriate primitive operations for manipulating these values, pre-defined in the type system. Base types are also called atomic types, because they have no internal structure as far as the type system is concerned. A *base*

[3] http://www.iso.ch/iso/en/prods-services/popstds/datesandtime.html

type in the temporal type system is a base type in this sense, and in the sense, that it is the reference type for any further type declared (w.∴include Definition 3), using type constructors of the type system.

The type system supports a single base type called *reference time partition type.* A reference time partition type is a set of *parts* of some chosen reference time partition whose parts are indexed by integers – plus appropriate primitive operations to manipulate these values. A *reference time partition* is a time partition of the time domain according to Definition 2 with a *fixed part* in the time domain indexed by 1. A reference time partition with a fixed part is required for a computer implementation of the multi-calendar temporal type system to reckon time: Fix an arbitrary starting point as 1 in the time domain, and specify any other part of this time partition by giving a number relative to that fixed point. The duration of any part of the reference time partition is 1.

The reference time partition type with a fixed part in the time domain provides a semantics for any further type declared (see below Section 3.2 for details). In particular, conversions to and from this reference type to other types become possible which can be expressed by mappings to and from some chosen reference time partition, of course with a fixed part anchored in the time domain. This enables processing of dates and times from arbitrary types and/or calendars and type casting. Note that the reference time partition must be chosen such that any further time partition may be declared directly or indirectly in terms of the reference time partition.

In the type system, the reference time partition type `Ref` has the constant `1` (intended to represent the fixed part of the reference time partition) and the operations `next` (successor part) and `previous` (predecessor part) to enumerate the parts of the reference time partition, and the duration of any part of type `Ref` is `1 Ref`.

For example, we may chose the time partition `second` as the reference time partition type `Ref` with midnight at the onset of Thursday, January 1 of year 1970 (Gregorian) as fixed point second 1 (according to Unix time). It is thus the first second of the first minute and the first hour; this assumption is correct[4]) and valid for any civil calendar in use today. The duration of a second in type `second` is `1 second`.

In the concrete syntax of the type system, i.e. the syntax used by some programmer, the reference time partition type is referenced by its name, e.g. `second`. A part (i.e. a *value* of the reference time partition type) is specified by its integer index (e.g. `second(2)`). In addition to the (integer) index (relative to the fixed point) for each part of this reference time partition, textual values to represent dates and times for input and output of the type's values are supported, as well. We suggest to choose the ISO Standard 8601 for Dates and Time[5] as textual representation format in the type system; then `second(1)` might equally be used with `'1970-01-01T00:00:00'`, for example.

[4] International System of Units (SI), http://www.bipm.org/en/si/
[5] http://www.iso.ch/iso/en/prods-services/popstds/datesandtime.html

3.2 Structured Types

This section introduces the different ways of building structured types in the multi-calendar temporal type system, using type constructors and selectors.

Time Partitions Many programs need to deal with time partitions whose parts are aggregations of sets of parts of the reference time partition. These time partitions can be (directly or indirectly) constructed from the reference time partition. The type mechanism that supports this kind of programming with arbitrary time partitions is *time partition type.* The elements of a time partition type are called *parts.*

A time partition is a data structure that specifies a partition of the time domain into finite or infinite parts (cf. Definition 2), possibly of different durations, by an aggregation of sets of parts of the reference time partition or another, already declared, time partition as follows: A time partition type specifies the first part, the durations, and an ordered finite periodic pattern of parts of different durations, possibly with finite many exceptions such that all parts of a time partition are clearly located by possibly infinite (convex) intervals in the time domain. The type constructor for time partition types is `timepartition`. For every time partition `T`, the type

`timepartition` d_1`[named` n_1`]>>` $\ldots$ `>>` d_m`[named` n_m`] where anchor=a`

describes a time partition `T` whose parts are constructed from an already declared time partition `S` such that time partition `T` *includes* time partition `S`, and `S` is *included in* `T` (cf. Definition 3). With the type constructor `timepartition`, $d_i, i \in \{1, ..., m\}$ are durations in an already declared time partition `S`, and `a`, the anchor is the first part of a set of parts of `S` determining the first part of the time partition `T`. The order of the durations of the parts of time partition `T` is syntactically committed by $>>$, connecting the anchored durations in the given periodic order, called *periodic pattern* which may be nested. Exceptions in the durations of the parts of some time partition (e.g. leap seconds in the specification of the time partition `minute`) are captured by a common `case` expression (which can be nested). Subsets of a time partition `T` may be named by `[named` n_i`]`, $i \in \{1...m\}$. Each named subset of `T` builds a subtype of `T` usable as any other type declared. An additional syntactic form of the type constructor `timepartition` is provided, where the parts in the periodic pattern of the time partition declared can be numbered:

`timepartition 1:`d_1`[named` n_1`]...m:`d_m`[named` n_m`] where anchor=a`

The fundamental property of time partitions is that `T(i)`, the i^{th} part of time partition `T` can be computed according to the specifications in the type declaration for `T` quickly for any value $i \in \mathbb{Z}$ at run time. i is called the *(integer) index* of the i^{th} part of the time partition `T` relative to its first part (i.e. its anchor) located in the time domain. Of course, as it is the case for the reference time partition type, each part of a time partition type has an additional textual value representation of dates and times for input and output of type values.

Let's consider a small example. We declare the time partition `teaching_term` (according to teaching terms at Bavarian universities where winter terms always begin in October) by an aggregation of the time partition `month` (of the Gregorian calendar), assuming that `month` is already declared, as follows:

```
type teaching_term = timepartition
                       1: 6 month named winterterm
                       2: 6 month named summerterm
                     anchored_at month(10);
```

With this type declaration, a part of type `teaching_term` has a duration of `1 teaching_term`, and the teaching_term indexed by 1 is defined by including 6 months starting with the month indexed by 10. The i^{th} part of the time partition `teaching_term` is `teaching_term(i)`. `winterterm` is a subtype of `teaching_term` that can be used in the same way as the type `teaching_term`.

Selectors So far, we have studied the type constructor for time partitions totally covering the time domain $(\mathcal{T},<_{\mathcal{T}})$ in terms of Definition 2. But people frequently use particular selected, gapped subsets of a partition like `Saturday` of the partition `day` or `the 8'o clock news` of the partition `hour`: `Saturday` is the infinite set of every 6^{th} part out of a period of 7 parts of `day` and `the 8'o clock news` is an infinite set of every 8^{th} part out of any period of 24 parts of `hour`.

Selectors construct subtypes of a type by selecting specific sets of its supertype. We accomplish this by formalizing the intuition that any type `S` constructed by a selector is more informative than the type `T`, `S` is constructed from. We say `S` is a *subtype* of `T`, written `S` $<:$ `T`[6], to mean that any part of `S` can safely be used in a context where a part of `T` is expected. For types $\mathtt{T}_1$, $\mathtt{T}_2$, ..., and $i \in \mathbb{Z}$, the following selectors are supported. The selectors are subsequently explained by examples (using time and calendar concepts of the Gregorian calendar).

Selector	**Syntactic From**	**Conditions**
select	`select` $\mathtt{T}_1$`(i) where <` `condition` `>`	
select_during	`select_during(i,`$\mathtt{T}_1$`,`$\mathtt{T}_2$`)`	$\mathtt{T}_1 \trianglelefteq \mathtt{T}_2$
select_overlaps	`select_overlaps(i,`$\mathtt{T}_1$`,`$\mathtt{T}_2$`)`	neither $\mathtt{T}_1 \trianglelefteq \mathtt{T}_2$ nor $\mathtt{T}_1 \trianglerighteq \mathtt{T}_2$
includes	$\mathtt{T}_1$ `includes` $\mathtt{T}_2$	$\mathtt{T}_1 \trianglerighteq \mathtt{T}_2$
included_in	$\mathtt{T}_1$ `included_in` $\mathtt{T}_2$	$\mathtt{T}_1 \trianglelefteq \mathtt{T}_2$
union	$\mathtt{T}_1$ `union` $\mathtt{T}_2$	$\mathtt{T}_1$,$\mathtt{T}_2$ $<:$ $\mathtt{T}_3$
intersects	$\mathtt{T}_1$ `intersects` $\mathtt{T}_2$	$\mathtt{T}_1$,$\mathtt{T}_2$ $<:$ $\mathtt{T}_3$
minus	$\mathtt{T}_1$ `minus` $\mathtt{T}_2$	$\mathtt{T}_1$,$\mathtt{T}_2$ $<:$ $\mathtt{T}_3$
join	$\mathtt{T}_1$ `join` $\mathtt{T}_2$	$\mathtt{T}_1$,$\mathtt{T}_2$ $<:$ $\mathtt{T}_3$
shift	`shift(`$\mathtt{T}_1$`,d)`	d :: duration of $\mathtt{T}_1$

With the selector `select`, specific sets of a type can be selected using constraints over the type's parts expressed by *conditions* the parts must satisfy, usable after the keyword `where`. For example, the subtype `winterterm` of the type `teaching_term`, i.e. always the first out of two teaching terms, can be declared as follows:

[6] Denoting *predicate subtyping* constraining the parts of `T` satisfying a *selector*.

```
(1) type winterterm = select teaching_term(i) where i mod 2==1;
```

This selector comes with an additional syntactic form which provides a rather textual representation to formulate rather simple conditions, e.g.

```
(2) type winterterm = select teaching_term
       anchored_at teaching_term(1) in_period 2 teaching_term;
```

Note that the syntactic form (2) is merely syntactic sugar for the form (1).

The selectors `select_during` and `select_overlaps` enable to select subsets of parts of a time partition T_1 by locating specific parts in another time partition T_2 where the parts of the time partition T_1 are either *during, start,* or *finish* [2] those of T_2 (i.e. T_1 is included in T_2) or might *overlap* [2] those of T_2 (i.e. T_1 is not included in T_2). For example,

```
type christmas_day = select_during(25,day,december);
type 1stweek_winterterm = select_overlaps(1,week,winterterm);
```

With these type declarations `christmas_day` is a subtype of `day`, locating always the 25^{th} part of type `day` in each part of type `december`, and `1stweek_winterterm` is a subtype of `week`, locating always the first part of `week` that possibly overlaps with the respective part of type `winterterm`. The `select_overlaps` selector is used, because the type `week` is not included in the type `winterterm`.

The selectors `includes` and `included_in` select included (resp. including) parts of type T_1 in type T_2. For example,

```
type christmas_week = week includes christmas_day;
```

specifies the set of all those weeks which include a Christmas Day.

The `shift` selector shifts each part of type `T` by a duration of type `Duration of T`. For example, the $4th$ Advent can be specified as the Sunday in the week before the week including Christmas Day.

```
type Advent4=shift(sunday included_in christmas_week),-7 day);
```

The three selectors `union`, `minus`, and `intersects` are the usual set-theoretic operations over types. They can be, for example, used to declare the following calendar types:

```
type weekend_days := sunday union saturday;
type weekday = day minus weekend;
type sunday&lastday_month = sunday intersects lastday_month;
```

The set of all weekend days is the union of all Saturdays and Sundays, any weekday is the difference of days and weekend days, and those Sundays which are also the last day of some month are straightforwardly declared by union, minus, and intersects, respectively.

To declare a type `weekend` instead of `weekend_days`, for example, whose parts have the length of two consecutive days, the selector `join` over types T_1 and T_2 both having the same supertype can be used, concatenating the parts of both types.

Durations Many programs dealing with time make use of means to measure time or specifying extends of temporal events in terms of *durations.* A duration is a signed integral number of parts in any type with known length but no specific starting or ending parts.

The type constructor `duration of T` describes for any type T a duration drawn from the type `T`. A duration value of type `duration of T` formed by a signed number and a type `T` is written `q T` where $q \in \mathbb{Q}$. For example `YearsOfStudy` has a duration of 9 teaching_terms (at a Bavarian university).

```
YearsOfStudy::duration of teaching_term = 9 teaching_term;
```

Note that type ascription is not require for duration types in the type system, because type inferencing is supported. For example, the duration of the variable `YearsOfStudy` may be declared without previous type ascription.

Time Intervals For many programs dealing with time it is useful to have a mechanism to specify somehow related parts in some time partition building particular *time intervals.* The type mechanism that supports this kind of programming with time intervals is *time interval type.* A time interval is a finite or infinite sequence of possibly non-connected parts of some time partition where the parts are ordered and pairwise disjoint. For example, my holidays in 2003 between 8 to 14 April and 29 July to 6 August is a time interval in time partition `day`.

The type constructor for time intervals is `timeinterval`. For every type `T`, the type `timeinterval of T` describes finite or infinite ordered sequences of possibly non-connected parts whose elements are drawn from `T`.

Time intervals can be constructed by the following syntactic forms, i.e. by the following value constructors.

- The *empty* time interval (with elements of type `T`) is written `[]::T`.
- A time interval constructed by its *ending points* $\texttt{t}_1$ and $\texttt{t}_2$ (both of type `T`), including all parts between $\texttt{t}_1$ and $\texttt{t}_2$ of `T`, is written $\texttt{[t}_1\texttt{..t}_2\texttt{]::T}$.
- A time interval constructed by a *duration* `q T` (of type `duration of T`) and an *anchor* `t` (of type `T`) is written `q T from t::T`, meaning that `q` consecutive parts of `T` are included in the time interval where `t` is the first of those parts.
- A possibly infinite time interval constructed by a *selector* `s` (cf. Section 3.2) (of type `T`), possibly an *anchor* $\texttt{t}_1$ and possibly an *endpoint* $\texttt{t}_2$ (both of type `T`) is written $\texttt{s [from t}_1\texttt{] [to t}_2\texttt{]::T}$, meaning that those parts of `T` satisfying `s` between a possible anchor $\texttt{t}_1$ and a possible endpoint $\texttt{t}_2$ are included in the constructed time interval.

Time intervals constructed in one of the previously introduced forms can be related by a *comma,* interpreted as concatenation time intervals.

Let's turn attention to the following example. We assume that the types `day`, `month`, and `sunday` (all of the Gregorian calendar) are already declared.

```
Holidays03 = [2003-04-08..2003-04-14],[2003-07-29..2003-08-06];
KickoffMeeting = 4 day from 2004-03-01;
ClubMeeting = select_during(2,sunday,month) from 2004-02-15;
```

The variable `MyHolidays03` is of type time interval of days as previously illustrated. `KickoffMeeting` is a time interval of four consecutive days starting with day `2004-03-01` (and ending with day `2004-03-04`). And `ClubMeeting` is a time interval including always the second Sunday of each month starting with day `2004-02-15`. As illustrated, type ascription is also not required for variables and constants of type `timeinterval`, because type inferencing is supported.

3.3 Modules

Types in the multi-calendar temporal type system are declared within a *module*. A module defines a scope for a finite set of type declarations belonging together. A module has the following syntactic form:

```
CALENDAR [qualified] MY_CALENDAR
  (* finite set of type declarations *)
END
```

The attribute `qualified` is optional. If used, the declarations made in this module are restricted to the scope of this module. If the module is imported, i.e.

```
CALENDAR ANOTHER_CALENDAR
  import MY_CALENDAR;
  (* finite set of type declarations *)
END
```

then declaration `d` made within the module `MY_CALENDAR` and used in the module `ANOTHER_CALENDAR` is accessed by `MY_CALENDAR.d`. Thus, the name `d` might also be used for a declaration within the module `ANOTHER_CALENDAR`. For example, if the Gregorian calendar is declared in a qualified module `GREGORIAN` containing a type `christmas_day` and the Julian calendar is declared in another qualified module `JULIAN` containing also the type `christmas_day`, then Julian and Gregorian Christmas can be used together within some program using the dot notation: `GREGORIAN.christmas_day`[7] and `JULIAN.christmas_day`[8], respectively. That means, qualified calendar modules provide a means for dealing with context-dependent temporal and calendric data.

3.4 Examples

Let us consider a detailed example on programming with the multi-calendar temporal type system. Example 1 is an exemplary declaration of the Gregorian calendar, and Example 2 is an exemplary declaration of the arithmetic Islamic calendar in which months follow a pattern set. In this example, the declarations of the Islamic calendar are based on those of the Gregorian calendar (i.e. the calendar declarations are directly related), to enable straightforward comparison and casting of Islamic and Gregorian calendar concepts in some program. This can be realized by declaring one time partition of the Islamic calendar in terms of

[7] 25 December
[8] 7 January

a time partition of the Gregorian calendar, and subsequently relating any further type declaration to this time partition and the system's fixed part in time. Note that the type system also supports the possibility to declare the Islamic calendar independent of the Gregorian calendar, both only with relation to the reference time partition type, for example.

For the following examples we assume that the time partition `second` is the pre-defined *reference time partition type* with midnight at the onset of Thursday, January 1 of year 1970 (Gregorian) (according to Unix time) as fixed part `second 1`. Furthermore, we assume that the following relations and a function are pre-defined such that they are usable in the subsequent examples: (1) `leapYear` is a pre-defined relation that returns `true` if a given Gregorian month falls into a Gregorian leap year, otherwise it returns `false`; (2) `islamicLeapYear` is a pre-defined relation that returns `true` if a given Islamic month falls into an Islamic leap year, otherwise it returns `false`; and (3) `sunset(Locale)` is a pre-defined function that returns the time in local mean time for a given `Locale`.

Example 1. **(Gregorian Calendar.)**

```
CALENDAR GREGORIAN
 type minute = timepartition 60 second anchored_at second(1);
 type hour = timepartition 60 minute anchored_at minute(1);
 type day = timepartition 24 hour anchored_at hour(1);
 type week = timepartition 7 day anchored_at day(-2); (* begin on Mondays *)
 type month = timepartition
               1:  31 day named january
               2:  case
                      leapYear  = 29 day named february
                    | otherwise = 28 day named february
                   end
               3:  31 day named march
               4:  30 day named april
               5:  31 day named may
               6:  30 day named june
               7:  31 day named july
               8:  31 day named august
               9:  30 day named september
               10: 31 day named october
               11: 30 day named november
               12: 31 day named december
              anchored_at day(1);
 type year = timepartition 12 month anchored_at month(1);
 with select day(i) where i mod 7 == j
  type monday    = j == 5
  type tuesday   = j == 6
  type wednesday = j == 0
  type thursday  = j == 1
  type friday    = j == 2
  type saturday  = j == 3
  type sunday    = j == 4
 end
END
```

The Gregorian calendar is declared straightforward using some of the previously introduced type constructors defining specific sets with the intended meaning. Exceptions are captured using `case` expressions which are usable within any type declaration, e.g. the declaration of type `month` to capture the irregularity of Gregorian months due to Gregorian leap year rules. In Example 1 it is demon-

strated how type declarations can be group using the `with ...end` construct, declaring weekdays, for example.

In real life minutes contain leap seconds from time to time. This phenomenon can also be expressed in the type system:

```
type minute = timepartition case
                                  minute(1051200)      = 70 second
                                | containsLeapSeconds  = 61 second
                                | otherwise            = 60 second
                               end
                    anchored_at second(1);
```

assuming that `containsLeapSeconds` is a pre-defined relation that returns `true` if a given minute contains leap seconds, otherwise it returns `false`.

Note that type declarations for the most important Christian and Orthodox holidays are straightforward when using the temporal type system, extending the program given in Example 1 only by a few lines.

Now let us consider declarations for the Islamic calendar.

Example 2. **(Islamic Calendar.)**

```
CALENDAR ISLAMIC
import GREGORIAN;
 type i_day = shift(day,distance(sunset(Locale),midnight));
 type i_week = timepartition 7 i_day anchored_at i_day(-1); (* begin on Sundays *)
 type i_month = timepartition
                 1:  30 i_day named muharram
                 2:  29 i_day named safar
                 3:  30 i_day named rabiI
                 4:  29 i_day named rabiII
                 5:  30 i_day named jumadaI
                 6:  29 i_day named jumadaII
                 7:  30 i_day named rajab
                 8:  29 i_day named sha‘ban
                 9:  30 i_day named ramadan
                 10: 29 i_day named shawwal
                 11: 30 i_day named dhu_al-qa‘da
                 12: case
                        islamicLeapYear = 30 i_day named dhu_al_hijja
                      | otherwise       = 29 i_day named dhu_al_hijja
                     end
               anchored_at i_day(-286);
 type i_year = timepartition 12 month anchored_at i_month(1);
 with select day(i) where i mod 7 == j
   type yaum_al-ahad     = j == 4          (* Sunday    *)
   type yaum_al-ithnayna = j == 5          (* Monday    *)
   type yaum_al-thalatha = j == 6          (* Tuesday   *)
   type yaum_al-arba‘a   = j == 0          (* Wednesday *)
   type yaum_al-hamis    = j == 1          (* Thursday  *)
   type yaum_al-jum‘a    = j == 2          (* Friday    *)
   type yaum_al-as-sabt  = j == 3          (* Saturday  *)
 end
END
```

In Example 2 we define an explicit relation between the Gregorian and the Islamic calendar. In the Islamic calendar, the chosen fixed part, i.e. midnight at the onset of Thursday, January 1 of year 1970 (Gregorian) corresponds to 22 October 1398 (Islamic). Note that an Islamic day begins at sunset of the previous Gregorian day and ends at sunset of the next Gregorian day. The sunset

depends on a specific location. However, Islamic days are also defined by a period of 24 hours. The aspect of sunset is only of interest if we talk of evening or for determining the beginning of Islamic holidays. An Islamic type `i_day` may thus simply be declared by a shift according to sunset of the Gregorian calendar type `day` as illustrated in Example 2. The other time partition of this calendar are then straightforward declared in the expected manner.

Note that type declarations also for the most important Islamic holidays are straightforward when using the temporal type system, extending the program given in Example 2 only by a few lines.

4 Perspective: Integration into the Web Query Language Xcerpt

With the multi-calendar temporal type system, we develop a type language for time and calendar concepts with which the programmer can declare concepts of his/her needs within a (intern or extern) *module,* seamlessly integrated into a host (query) language. For proof-of-concept purposes, we have chosen the (Semantic) Web query language Xcerpt [12] as host language for the type system. The multi-calendar temporal type system integrated into the (Semantic) Web query language Xcerpt is intended to support temporal *adaptive Web systems*[9], formulating queries independent of a particular web-based application and/or a particular temporal or calendric context.

For explicitly ascribing a particular type to an Xcerpt term, we write `"t::T"` for "the term t which we ascribe the type T". A value or object `v` of some type defined in a module can be used as an Xcerpt term appearing in an Xcerpt program simply by using its name `v`.

Let us consider an oversimpled database (Example 3) of a travel agency containing air journey offers during the summer period as it is defined for the traveling industry (a type declaration for type `summer` is illustrated in the calendar `JOURNEY` used by some travel agency). The offers come with `destination, price-per-week,` and `bookable` elements. A person inquiries the travel agency for a one week journey during his summer vacations, which he/she has defined in the module `JOURNEY`, as well in the element `MyVacation04` (of type `timeinterval of day`). The module in Example 3 imports the Gregorian calendar module declared in Example 1 such that its declarations can be used in the module `JOURNEY`.

Example 3. **(Data Term and Query).**

```
CALENDAR JOURNEYS
   import_calendar GREGORIAN;
   type summer = june join july join august join september;
   Summer2004 = 2004 includes summer;
   MyVacation04 = [2004-07-19..2004-08-22];
END
```

[9] Adaptation basically means delivering and/or rendering data in a context-aware manner, i.e. combining parts of Web data depending on context specified, e.g. by some user model, parameters of the rendering, or time and location of some user.

```
air_journeys {                          }
 season { Summer2004 }
 journey {                                  CONSTRUCT
   destination { "Santorini" },               destinations { all var Dest }
   price-per-week { "329 €" },              FROM
   bookable { [2004-06-01..2004-06-20] or     air_journeys {{
              [2004-07-17..2004-08-04] or       journey {{
              [2004-09-15..2004-09-22] }           var Dest ⇝ destination,
 },                                                bookable { var Booking }
 journey {                                       }}
   destination { "Sicily" },                   }} where {
   price-per-week { "461 €" },                    zip_some_contains(var Booking,
   bookable { [2004-06-24..2004-07-23] or          (MyVacation04 includes week))
              [2004-08-02..2004-08-19] }          }
 }, ...                                     END
```

With Example 3, the tourist looks for a week which is included in his/her summer vacations in year 2004. This is expressed by the function `includes` casting the time interval `MyVacation04` to a time interval of weeks of the same time period. At least one of these weeks must be contained in one of the bookable time intervals. This is expressed with the relation `zip_some_contains`, generalizing Allen's interval relation *contains* [2] such that each of the bookable time intervals are related to each of the weeks according to the contains relation, returning true if at least one week is contained in one of the bookable time intervals.

Note that several functions and relations over values of any type are supported, e.g. Allen's interval relations [2], casting functions, and functions to relate parts and/or time intervals and durations. Relations over types can be used in an Xcerpt condition box (i.e. the WHERE-part of an Xcerpt query) and functions over temporal and calendric types supported with the temporal type system can be used within relations and in an Xcerpt CONSTRUCT-part.

5 Conclusion

This article has introduced the type constructors and selectors of a multi-calendar temporal type system seamlessly integrated into a host (query) language. Its associated type checking methods have not been considered in this article.

The type constructors and selectors of the type system presented in this article, form a small and simple, but powerful set: As illustrated in this article, different calendars like the Gregorian and the Islamic calendar can be defined; the Hebrew calendar, which is slightly more complicated than the addressed ones, may also be defined in the temporal type system. Furthermore, several time and calendar notions used in a university context and in a business context have been defined in the type system. For proof-of-concept purposes, we currently implement the introduced type constructors and selectors and several functions and operations over temporal types in the (Semantic) Web query language Xcerpt.

The type constructors and selector of the temporal type system, presented in this article, are designed to provide a means for modeling typed time and calendar data in (Semantic) Web queries in a declarative way integrated into its host language, to allow for context-aware queries, and to enable static type checking. A type amenable to static type checking, i.e. at compile time before

the actual value in some program is computed, must be 'value independent'. For this reason, a type like `working_week` which depends on concrete values like the US holiday Independence Day being in each year at $4th$ July must be derived from a more general time partition type. Note that other time models for calendars are data dependent (e.g. [6, 7]). Type checking and type inferencing types in the multi-calendar temporal type system is an interesting challenge due to specific aspects of time concepts and calendar systems. For example, type equivalence since a type like `4thAdvent` might be declared in different ways, however defining the same time, or type casting, because casting type `week` to type `month` information is lost that cannot be recovered. In addition, two interesting typing relations *includes* and *subtype of* between temporal and calendric types exist. Beyond this, several relations and functions supported over temporal and calendric types are *polymorph.*

References

1. McDermott, D.V.: A temporal Logic for Reasoning about Processes and Plans. Cognitive Science **6** (1982) 101–155
2. Allen, J.F.: Maintaining Knowledge about temporal Intervals. Communications of the ACM **26** (1983) 832–843
3. van Benthem, J.: The Logic of Time. Studies in Epistemology, Logic, Methodology, and Philosophy of Science. D. Reidel Publishing Company (1983)
4. Gabbay, D.M., Hodkinson, I., Reynolds, M.: Temporal Logic. Mathematical Foundations and Computational Aspects, Vol. 1. Oxford University Press Inc. New York (1994)
5. Vila, L.: A Survey on temporal Reasoning in Artificial Intelligence. Artificial Intelligence **7** (1994) 4–28
6. Chandra, R., Segev, A., Stonebraker, M.: Implementing Calendars and temporal Rules in next Generation Databases. In: Proc. Int. Conf. on Data Engineering. (1994) 264–273
7. Bettini, C., Jajodia, S., Wang, X.S.: Time Granularities in Databases, Data Mining, and temporal Reasoning. Springer Verlag, Berlin (2000)
8. Ning, P., Wang, X.S., Jajodia, S.: An Algebraic Representation of Calendars. In: the Annuals of Mathematics and Artificial Intelligence (Kluwer), to appear. (2001)
9. Dershowitz, N., Reingold, E.M.: Calendrical Calculations: The Millennium Edition. Cambridge University Press (2001)
10. Ohlbach, H.J.: WebCal, an advanced Calendar Server. In: Technical Report. University of Munich. (2003)
11. Bry, F., Lorenz, B., Ohlbach, H.J., Spranger, S.: On Reasoning on Time and Location on the Web. In: Proc. Workshop on Principles and Practice of Semantic Web Reasoning, LNCS 2901, Springer-Verlag. (2003)
12. Schaffert, S., Bry, F.: Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In: Technical Report, PMS-FB-2004-7. University of Munich. (2004)
13. Bry, F., Spranger, S.: Temporal Constructs for a Web Language. In: Proc. 4^{th} Workshop on Interval Temporal Logics and Duration Calculi (ESSLLI). (2003)