CaTTS: Calendar Types and Constraints for Web Applications

François Bry
University of Munich, Germany
bry@pms.ifi.lmu.de

Frank-André Rieß
University of Munich, Germany
riess@pms.ifi.lmu.de

Stephanie Spranger
University of Munich, Germany
spranger@pms.ifi.lmu.de

ABSTRACT

Data referring to cultural calendars such as the widespread Gregorian dates but also dates after the Chinese, Hebrew, or Islamic calendars as well as data referring to professional calendars like fiscal years or teaching terms are omnipresent on the Web. Formalisms such as XML Schema have acknowledged this by offering a rather extensive set of Gregorian dates and times as basic data types. This article introduces into CaTTS, the Calendar and Time Type System. CaTTS goes far beyond predefined date and time types after the Gregorian calendar as supported by XML Schema. CaTTS first gives rise to *declaratively specify* more or less complex cultural or professional calendars including specificities such as leap seconds, leap years, and time zones. CaTTS further offers a tool for the *static type checking* (of data typed after calendar(s) defined in CaTTS). CaTTS finally offers a language for *declaratively expressing* and a solver for *efficiently solving* temporal constraints (referring to calendar(s) expressed in CaTTS). CaTTS complements data modeling and reasoning methods designed for *generic* Semantic Web applications such as RDF or OWL with methods *specific* to the particular application domain of calendars and time.

Categories and Subject Descriptors

I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods—*representation languages*; D.3.1 [**Programming Languages**]: Formal Definition and Theory—*syntax*

General Terms

Languages

Keywords

time, calendars, types, Web reasoning

1. INTRODUCTION

Data referring to cultural calendars such as the widespread Gregorian dates but also dates after the Chinese, Hebrew, or Islamic calendars as well as data referring to professional calendars like fiscal years or teaching terms are omnipresent on the Web. Most likely, they will play an essential role on the Semantic Web, too. Formalisms such as XML Schema have acknowledged this by offering a rather extensive set of Gregorian dates and times as basic data types.

WWW2005, May 10–14, 2005, Chiba, Japan.

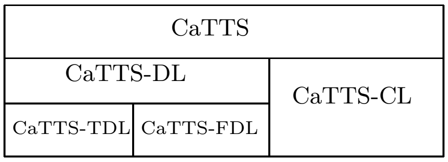


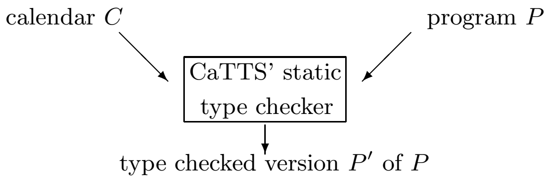
Figure 1: Languages of CATTS.

This article introduces into CaTTS, the Calendar and Time Type System that goes far beyond predefined date and time types after the Gregorian calendar. CaTTS consists of two languages, a *type definition language*, CaTTS-DL, and a *constraint language*, CaTTS-CL, of a (common) parser for both languages, and of a language processor for each language.

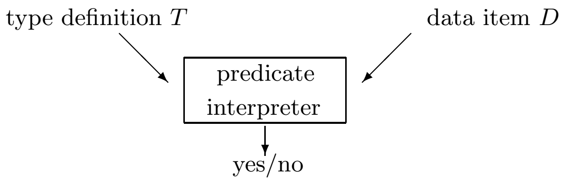
Using the (type) definition language CaTTS-DL, one can specify in a rather simple manner more or less complex, cultural or professional calendars [3]. Specificities like leap seconds, leap years, and time zones can be easily expressed in CaTTS-DL. Calendars expressed in CaTTS-DL are composable in the sense that the language offers a means for modules. Thus, one can extend a standard calendar such as the Gregorian calendar used in Germany with a particular teaching calendar, e.g. the one of a specific German university.

CaTTS-DL gives rise to defining both, *calendric data types* specific to a particular calendar – such as "working day", "Easter Monday", "exam week", or "CS123 lecture" (defining the times when the Computer Science lecture number 123 takes place) – using the language fragment CaTTS-TDL (for Type Definition Language) of CaTTS-DL, and *data formats* for such data types – such as "5.10.2004", "2004/10/05", or "Tue Oct 5 16:39:36 CEST 2004" – using the language fragment CaTTS-FDL (for Format Definition Language) of CaTTS-DL.

The language processor for CaTTS-DL consists of two software components, a *static type checker* and a *predicate interpreter*. Consider a calendar C specified in CaTTS-DL and a program P in the language CaTTS-CL, XQuery, XSLT or any other program P with type annotations referring to types and/or formats specified in the calendar C. CaTTS' static type checker verifies and/or extends the type annotations in P generating a type checked version P' of P:



Consider a type definition T (e.g. "working day" or "exam week") from a calendar specified in CaTTS-DL and a data item D (e.g. "2004/10/05"). CaTTS' predicate interpreter detects whether the data item D has type T (e.g. whether "2004/10/05" is a "working day" or an "exam week" - the later being probably false):



Both CaTTS-DL and CaTTS-CL provide (the same) predefined functions (e.g. shift forward and backward, during, and usual arithmetics). Using these functions, one can express for example, "the second day after working day X" (in CaTTS: `shift X:working_day forward 2 day`).

CaTTS' type checker can be used for static type checking of programs or specifications in *any* language (e.g. XQuery, XSLT, XML Schema), using date formats enriched with type annotations after some calendar specified in CaTTS-DL. In particular, it is used for the static type checking of temporal constraint programs in CaTTS-CL, the constraint language of CaTTS.

Using CaTTS' constraint language CaTTS-CL, one can express a wide range of temporal constraints referring to the types defined in calendar(s) specified in the definition language CaTTS-DL. For example, if one specifies in CaTTS-DL a calendar defining both, the Gregorian calendar (with types such as "Easter Monday" or "holiday") and the teaching calendar of a given university (with types such as "working day", "CS123 lecture", and "exam week"), then one can refer in CaTTS-CL to "days that are neither holidays, nor days within an examination week" and express constraints on such days such as "strictly after Easter Monday and before June". Thus, using CaTTS-CL one can express real-life, Web, Semantic Web, and Web service related problems such as searching for train connections or making appointments (e.g. for audio or video conferences) over several time zones.

CaTTS provides with a constraint solver for problems expressed in CaTTS-CL. This solver refers to and relies on the type predicates generated from a calendar definition in CaTTS-DL. This makes search space restrictions possible that would not be possible if the calendar and temporal notion would be specified in a generic formalism such as first-order logic and processed with generic reasoning methods such as first-order logic theorem provers.

2. CATTS AND THE SEMANTIC WEB

CaTTS complements data modeling and reasoning methods such as RDF [15] or OWL [14] designed for *generic* Semantic Web applications with methods *specific* to a particular application domain, that of calendars and time. CaTTS approach is a form of "theory reasoning" like "paramodulation". Like paramodulation ensures natural expressing and efficient processing of equality in resolution theorem proving, CaTTS makes a user friendly expression and an efficient processing of calendric types and constraints possible. CaTTS departs from time ontologies such as the DAML Ontology of Time [4] or time in OWL-S [10] as follows:

- CaTTS considerably simplifies the modeling of specificities of cultural calendars (such as leap years, sun-based cycles like Gregorian years, or lunar-based cycles like Hebrew months),
- CaTTS provides both, a static type checker and type predicates for every CaTTS calendar(s) specification,
- CaTTS comes along with a constraint solver dedicated to calendar definitions that (1) processes CaTTS type annotations as constraints and (2) the language of which is amenable to CaTTS static type checking.

3. CATTS' MODEL IN A NUTSHELL

This section is a mathematical prologue that can be skipped in a first reading.

CaTTS' notion of time is *linear*. CaTTS is not intended for expressing possible futures, hence it is not based on a "branching time". Most common-sense, Web and Semantic Web and many Web service application can be conveniently modeled in a linear time framework.

CaTTS' notion of time is purely *interval-based*, i.e. temporal data of every kind have a duration. This reflects a widespread common-sense understanding of time according to which one mostly refer to time interval, not to time points. E.g. one refers to "2004/10/05", a day, or to "1st week of October 2004", a week. Even time point-like data such as 9:25 can be perceived as having a duration, possibly as small as one second or one millisecond. Considering only time intervals and no time points has two advantages. First, it significantly simplifies data modeling, an advantage for CaTTS' users. Second, it simplifies data processing, i.e. static type checking and constraint reasoning, an advantage for CaTTS' language processors. However, CaTTS can deal with time point-like data like the beginning of a week or whether a day d falls into a week w or not, as well.

In order to *formalize* CaTTS (time point-less) model, however, time points have to be considered:

Definition 1. A **base time line** is a pair $(\mathcal{T}, <_\mathcal{T})$ where $\mathcal{T}$ is an infinite set (isomorphic to $\mathbb{R}$) and $<_\mathcal{T}$ is a total order on $\mathcal{T}$ such that $\mathcal{T}$ is not bounded for $<_\mathcal{T}$. An element t of $\mathcal{T}$ is called **time point**.

Each CaTTS type definition creates a *discrete* image of the time line since CaTTS data types define data items with durations. E.g. data types "day" and "working day" imply two (different) images of the time line: days partition the time line, working days correspond to a portion of the day partition of the time line. Both images of the time line are discrete, i.e. isomorphic to $\mathbb{Z}$. The notion of *time granularity* [2] formalizes such "discretizations" of the time line:

Definition 2. Let $(\mathcal{T}, <_\mathcal{T})$ be a base time line. Let $G = \{g_i \mid i \in \mathbb{Z}\}$ be a set isomorphic to $\mathbb{Z}$. Let's call the elements of G **granules**. A **time granularity** is a (non-necessarily total) function $\mathcal{G}$ from G in the set of intervals over $\mathcal{T}$ such that for all $i, j \in \mathbb{Z}$ with $i < j$

1. if $\mathcal{G}(g_i) \neq \emptyset$ and $\mathcal{G}(g_j) \neq \emptyset$, then for all $t_i \in \mathcal{G}(g_i)$ and for all $t_j \in \mathcal{G}(g_j)$ $t_i <_\mathcal{T} t_j$.
2. If $\mathcal{G}(g_i) = \emptyset$, then $\mathcal{G}(g_j) = \emptyset$.

Examples of granules are days, working days, weeks, months, holidays, etc. According to Definition 2, two different granules of a same time granularity do not overlap. The first condition of Definition 2 induces from the ordering of the time points (of the base time line) the common-sense ordering on granules: e.g. the day "10/25/2004" is after the day "10/24/2004". The second condition of Definition 2 is purely technical: it makes it possible to refer to the *infinite* set $\mathbb{Z}$ also for *finite* sets of granules (e.g. someone's exam days during his/her years of study).

Definition 3. Let $\mathcal{G}$ be a time granularity and $g_i, g_j \in \mathcal{G}$ granules.
A **duration** D **over** $\mathcal{G}$ is a number (expressed as an unsigned integer) of granules of $\mathcal{G}$.

Thus, a duration can be informally understood as a time interval over a granularity with a given length but with no specific starting or ending point.

Subtypes are defined in CaTTS in terms of either *inclusion* or *aggregation* of time granularities. E.g. a type "working day" is an inclusion (in the common set-theoretic sense) of the type "day" since the set of working days is a subset of the set of days; the type "week" is an aggregation (in constructive set-theory) of the type "day" since each week can be defined as a time interval of days.

Definition 4. Let $\mathcal{G}$ and $\mathcal{H}$ be time granularities. $\mathcal{G}$ is an **aggregation subtype of** $\mathcal{H}$, denoted $\mathcal{G} \preceq \mathcal{H}$, if every granule of $\mathcal{G}$ is an interval over $\mathcal{H}$ and every granule of $\mathcal{H}$ is included in (exactly) one granule of $\mathcal{G}$.

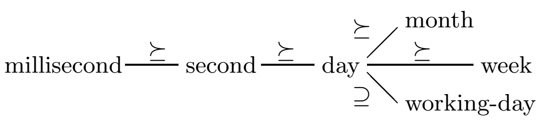
Definition 5. Let $\mathcal{G}$ and $\mathcal{H}$ be time granularities. $\mathcal{G}$ is an **inclusion subtype of** $\mathcal{H}$, denoted $\mathcal{G} \subseteq \mathcal{H}$ if every granule of $\mathcal{G}$ is a granule of $\mathcal{H}$.

The two subtype relations, inclusion subtype and aggregation subtype, are corner stones of CaTTS that, to the best of the knowledge of the authors, have not been proposed elsewhere. As the examples given below show, they are very useful in modeling calendars. Indeed, they reflect widespread forms of common-sense reasoning with calendric data.

Note that $\preceq \cup \subseteq$ is an order relation on time granularities. $\preceq \cup \subseteq$ is defined as follows: $\mathcal{G}_1 \preceq \cup \subseteq \mathcal{G}_2$ iff $\mathcal{G}_1 \preceq \mathcal{G}_2$ or $\mathcal{G}_1 \subseteq \mathcal{G}_2$. The following formalization of the notion of "calendar" reflects the central role played by aggregation and inclusion subtyping in CaTTS:

Definition 6. A **calendar** $C = \{\mathcal{G}_1, \ldots \mathcal{G}_n\}$ is a finite set of time granularities such that there exists a $\mathcal{G}_i \in C$ and for all $\mathcal{G}_j \in C$, $i, j \in \{1...n\}$ and $i \neq j$ $\mathcal{G}_j$ is $\preceq \cup \subseteq$-comparable with $\mathcal{G}_i$.

The subsequently illustrated set of time granularities defines a calendar; each of the time granularities is $\preceq \cup \subseteq$-comparable with the time granularity "second".



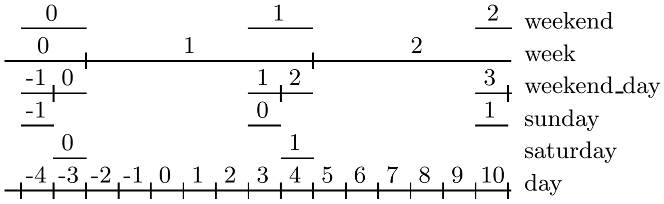


Figure 2: Indexing of the types defined in Section 4 (day(1) is Thursday, January 1, 1970).

4. CATTS-DL: DEFINITION LANGUAGE

Recall that CaTTS' definition language, CaTTS-DL, consists of a type definition language, CaTTS-TDL, and a date format definition language, CaTTS-FDL.

CaTTS-TDL provides a set of type constructors for declaring time granularities as subtypes (of predefined or user-defined) types in terms of *predicates,* called *predicate subtypes.* A subtype in CaTTS specifies (in terms of a predicate) a set, referred to as "predicate set". The usual set-theoretic operations (e.g. $\cup$) can be applied to predicate sets. In CaTTS, calendars (cf. Definition 6) are themselves typed by *calendar signatures.* This makes CaTTS calendar specifications reusable, maintainable, and easy to extend. In addition, user-definable *calendar functions* can be specified in CaTTS that parameterize calendars. Such functions provide a means to specify different calendar versions all having the same calendar signature. CaTTS-FDL provides a means to specify specific constants, i.e. the data items of predicate subtypes defined in a CaTTS-TDL calendar definition.

4.1 Reference Time

Each CaTTS implementation has a single predefined (base) type called `reference`.[1] `reference` is a time granularity such as "second" or "hour", chosen e.g. depending on the operating system. If the type `reference` is used in a CaTTS-DL calendar specification, then all other (user-defined) types are expressed directly or indirectly in terms of `reference`, using CaTTS' aggregation and/or inclusion subtyping (cf. Section 3). Note that the use of `reference` is optional in some CaTTS-DL calendar specification. If `reference` is e.g. the time granularity "second", a convenient choice with the Unix operating system, then one can specify (using CaTTS-DL) further coarser time granularities such as "day", "week", and "year" as well as finer time granularities such as "millisecond". The reference type makes conversions among any other types defined in a CaTTS-DL calendar specification (in terms of aggregation and/or inclusion), and, thus, between different CaTTS-DL calendars possible.

In CaTTS' prototype implementation, `reference` is the time granularity "second" of Unix (UTC seconds with midnight at the onset of Thursday, January 1 of year 1970 (Gregorian) as fixed point indexed by 1).

In CaTTS-DL, one defines a type finer than that of `reference` as follows:

```
type millisecond = refinement 1000 @ reference(1);
```
[2]

The type (time granularity) `millisecond` is defined as a thousandth refinement of a second (recall `reference` is the

[1] `reference` is a base type, because it has no internal structure as far as the type system of CaTTS is concerned.

[2] In this and the following examples, type identifiers start with lower case letters.

time granularity "second"). The type `millisecond` is anchored at (denoted @) second 1 (denoted `reference(1)`), since `millisecond(1)` is the first millisecond in the interval of milliseconds specifying the first second (i.e. `reference(1)`). Thus, in CaTTS types such as `reference` or `millisecond` induces clear integer indices of its data items. This indexing of data items turns out to be extremely useful in practice (cf. below).

4.2 Predicate Subtypes

Infinite sets are logically encoded by predicates: for any set A, the predicate $p : A \to \mathbb{B}$ defines the set of those elements of A that satisfy p. Such sets are called *predicate sets.* In type theory, predicate sets are interpreted as *predicate subtypes*, particularly used to express dependent function types [11]. CaTTS uses predicate subtypes in a different manner and not for theoretical, but instead practical purposes.

CaTTS uses predicate subtypes as a means to define calendric types. E.g. one can describe the time granularity "week" as the *subset* of those time intervals over the time granularity "day" having a duration of 7 days and beginning on Mondays. This can be directly expressed in CaTTS-DL as follows:

```
type week = aggregate 7 day @ day(-2);
```

The type `week` is an aggregation of days such that each data item is an interval with duration "7 days". The first week (i.e. `week(1)`) is anchored at (denoted @) `day(-2)`, i.e. the first day of the interval of days *aggregated* to the data item `week(1)`. Since `day(1)` is a Thursday (recall that CaTTS' implements Unix time with Thursday, January 1, 1970 as fixed point indexed by 1), `day(-2)` is a Monday. Any further index can be computed relative to this anchor (cf. Figure 2). Thus, the predicate subtype `week:day` $\to \mathbb{B}$ specifies the infinite set of those day time intervals satisfying the *predicate* "week" as previously expressed in CaTTS-DL. The type `week` is an *aggregation subtype* of the type `day` (cf. Definition 4), written `week <: day` in CaTTS' syntax, and read as "week is an aggregation of day". In CaTTS-DL, aggregation subtypes are constructed by the `aggregate` constructor, or by the predicates "#<" ("coarser restriction") and "#>" ("finer restriction"), set-based operations defined for CaTTS due to predicative set aggregation. E.g. aggregating the type `weekend_day` into the type `week` yielding the type `weekend`, a coarser restriction of `week` to `weekend_day`. The two set-based aggregation predicates may also be used to define types having non-convex data items like "working_week" (a coarser restriction of working days, i.e. days that are neither weekend days nor holidays, into weeks). Note that one might define aggregations having data items of different durations involving often complex conditions (e.g. Gregorian or Hebrew months), as well.

One can describe the time granularity "weekend day" as the *subset* of those granules of time granularity "day" that are either Saturdays or Sundays. This can be directly expressed in CaTTS-DL as follows:

```
type saturday = select day(i) where
                  relative i in week == 6;
type sunday =   select day(i) where
                  relative i in week == 7;
type weekend_day = saturday | sunday;
```

The type `saturday` is a selection of the 6^{th} day relative in each week. Thus, the predicate subtype `saturday:day` $\to \mathbb{B}$ specifies the infinite set of those days satisfying the *predicate* "saturday" as previously expressed in CaTTS-DL. In the same manner, the type `sunday` is defined. Finally, the type `weekend_day` satisfies either the predicate of type `saturday` *or* (denoted "|") the predicate of type `sunday`. The types `saturday, sunday,` and `weekend_day` are *inclusion subtypes* of the type `day` (cf. Definition 5), written e.g. `weekend_day` c : `day` in CaTTS' syntax, and read as "weekend day is an inclusion of day". In CaTTS-DL, inclusion subtypes are constructed by the `select` constructor by specifying a predicate as a combination of any operation predefined in CaTTS (cf. Appendix A for the complete syntax of CaTTS) following the reserved word `where,` or by a set-based constructor: "|" ("or"), "&" ("and"), and "\" ("except"), in fact ordinary set operations interpreted as predicate subtype constructors in CaTTS-DL. As illustrated in Figure 2, the indexing of inclusion subtypes is successive. The indexing of any inclusion subtype is however implicitly related to that of its inclusion supertype, appropriately choosing the data item indexed by 0 relative to that one of the supertype.

```
calendar type STD =
 sig
   type second;
   type minute        <: second;
   type hour          <: minute;
   type day           <: hour;
   type week          <: day;
   type month         <: day;
   type year          <: month;
   group day_of_week  c: day;
   type weekend_day   c: day;
   group holiday      c: day;
 end
```

Figure 3: Signature of a standard calendar in CaTTS-DL.

Note that weekend days may alternatively be defined as a `group,` i.e. a named collection of type definitions in CaTTS-DL:

```
group weekend_day =
  with select day(i) where (relative i in week)==j
     type saturday where j == 6
     type sunday   where j == 7
  end
```

Having defined the group `weekend_day,` one can either referring to data items of the group `weekend_day,` or, more specific, to data items of one of the types (`saturday, sunday`) defined within the group, explicitly defined belonging to a less specific "kind", i.e. a group of a CaTTS' type.

Furthermore, CaTTS provides a *polytypic constructor:* $|\tau|$ (durations of τ) that describes for any type τ a duration (cf. Definition 3) whose data items are drawn from τ. For example, |`day`| is the type of durations of days.

Note that CaTTS' type constructors provide a natural way to obtain *periodic events,* e.g. the CS123 lecture offered within the summer term 2004 (referred to as `ST04`) (in CaTTS `X:CS123_lecture during ST04`) of type **`CS123_lecture`**. Note that periodic events are very frequently considered in current research. The authors believe that CaTTS offers a particularly convenient and intuitive manner to specify periodic events.

4.3 Calendar as Type

The basic entities of CaTTS-DL calendar definitions (cf.

Definition 6) are *calendars, calendar signatures,* and *calendar functions.*

4.3.1 Calendars

A calendar is a packaged, finite collection of CaTTS type definitions and calendar specifications, assigning types to type identifiers, groups to group identifiers, and calendars to calendar identifiers, similar to ML structures [9] or XML documents [13]. The types and calendars specified in a calendar are delimited by the keywords `cal` and `end`. The following specification binds a calendar to the identifier `Cal`. This calendar defines an environment mapping `day, weekend_day, week,` and `weekend` to their respective group and type definitions.

```
calendar Cal =
 cal
  type day;
  group weekend_day = with select day(i) where
    (relative i in week) == j
      type saturday where j == 6
      type sunday   where j == 7
    end
  type week = aggregate 7 day @ day(0);
  type weekend = week #< weekend_day;
 end
```

Note that `day` is a user-defined reference type (isomorphic to the integers) without any further specifications. The identifiers in a calendar are qualified. For example, the qualified identifier `Cal.week` refers to the component `week` in the calendar definition `Cal.`

4.3.2 Calendar Signatures

Calendar signatures are kinds of "types" for calendars defined in CaTTS-DL, similar to ML [9] signatures or XML Schema declarations [12]. Calendar signatures specify identifiers and abstract predicate subtypes in terms of inclusion or aggregation supertypes for each of the components of a calendar implementing the signature. The following specification binds a calendar signature to the identifier `SIG.`

```
calendar type SIG =
 sig
   type day;
   group weekend_day c: day;
   type week <: day;
   type weekend c: weekend_day;
 end
```

This calendar signature describes those calendars having a type `day,` a group `weekend_day`, where each type belonging to must be an inclusion subtype of `day` and types `week` as aggregation subtype of `day` and `weekend` as aggregation subtype of `weekend_day`. Since the calendar `Cal` introduced above satisfies this calendar signature, it is said to *match* the calendar signature `SIG.` Note that a defined calendar usually matches more than one calendar signature, and vice versa, a calendar signature may be implemented by more than one calendar. Calendar signatures in CaTTS-DL may be used to define *views* of calendars due to *ascription,* i.e. specifying less components than implemented in any matching calendar. The non-specified components are thus local to its calendar specification.

4.3.3 Calendar Functions

Calendar functions are user-defined functions on calendars using a syntax similar to function declarations in many programming languages. A calendar function `CF` defining Hebrew weekend days can be declared in CaTTS as follows:

```
cal_fun CF(C:SIG):SIG =
 cal
   group weekend_day = with select C.day(i)
     where (relative i in C.week) == j
           type friday   where j == 5
           type saturday where j == 6
     end
 end
```

The calendar function `CF` takes as argument any calendar `C` matching the calendar signature `SIG,` and yields as result a calendar also matching `SIG.` When applied to a suitable calendar, the calendar function `CF` yields as result the calendar whose group `weekend_day` is that of (Hebrew) weekend days, i.e. Fridays and Saturdays. Furthermore, *any* type definition in `C` depending on that of `weekend_day` is changed respectively to the calendar function `CF` when applied to C. E.g. applying `CF` to the previously illustrated calendar definition `Cal` yields a "new" group `weekend_day` and the type `weekend` is changed, as well. Since the calendar function `CF` may be applied to *any* calendar matching the signature `SIG,` the function is polymorph, thus define a parameterized calendar.

4.4 CaTTS-FDL

With most applications, one would appreciate not to specify dates and times using indices of the data items of CaTTS types like `day(23)` or `second(-123)`, but instead *date formats* like "5.10.2004", "2004/10/05", or "Tue Oct 5 16:39:36 CEST 2004". CaTTS-FDL provides a means for defining date formats.

A CaTTS-FDL date format is defined by a sequence of string constants and/or placeholders, the latter appearing as variable names optionally followed by a representation descriptor[3]. Consider the following example:

```
format stdfmt:day = y "-" m "-" d where
  stdfmt within year(y),
  stdfmt within M:month,
       m == relative index M      in year,
       d == relative index stdfmt in month;
```

This CaTTS-FDL format specification binds the identifier `stdfmt` to a standard format for data items of type `day`. Both a parser and a render function (i.e. *String* → `day` as well as `day` → *String*) can be computed by resolving the constraints associated with `stdfmt`. Given "1970-1-1", we get that `y` = 1970, `m` = 1 and `d` = 1 and can thus solve the constraints to get e.g. `Gregorian.day(1)`. Vice versa, given `(Gregorian.)day(1)`, we find that it is within `year(1970)`, within `M`, which is equal to `month(1)` and is the 1^{st} in a year, and that `day(1)` is the 1^{st} in a month, yielding the same values for `y`, `m` and `d` as above.

Catalogs are collections of formats specifying to which calendar signature these formats can be applied. Such catalogs may be nested, applying to common scoping rules.

```
catalog ISO: STD =
  cat
(* nested catalog for extended ISO formats *)
  catalog Extended =
    cat
      format stdfmt:day = y "-" m "-" d where
        stdfmt within year(y),
        stdfmt within M:month,
             m == relative index M      in year,
             d == relative index stdfmt in month;
    end
  end
```

[3] If none is given, that portion of the format is interpreted as a variable-length unsigned integer.

```
import LeapSeconds;
calendar Gregorian:STD =
 cal
   type second = reference;
   type minute = aggregate minute (i)
     alternate
       | i == 1051200 (*1.1.1972*) -> 70 second
       | hasLeapSec?(i)            -> 61 second
       | otherwise                 -> 60 second
     end @ second(1);
   type hour = aggregate 60 minute @ minute(1);
   type day = aggregate 24 hour @ hour(1);
   type week = aggregate 7 day @ day(-2);
   type month = aggregate month(i)
     31 day named january,
     alternate
      |((i div 12)-1970) mod 4 == 0 &&
        (((i div 12)-1970) mod 100 != 0 ||
         ((i div 12)-1970) mod 400 == 0) -> 29 day
      | otherwise                         -> 28 day
     end named february,
     31 day named march,
     30 day named april,
     31 day named may,
     30 day named june,
     31 day named july,
     31 day named august,
     30 day named september,
     31 day named october,
     30 day named november,
     31 day named december @ day(1);
   type year = aggregate 12 month @ month(1);
   group day_of_week =
     with select day(i) where
      (relative i in week) == j
        type monday    where j == 1
        type tuesday   where j == 2
        type wednesday where j == 3
        type thursday  where j == 4
        type friday    where j == 5
        type saturday  where j == 6
        type sunday    where j == 7
     end
   type weekend_day = saturday | sunday;
   group holiday = with select day(i) where
    (relative i in M) == j
     type all_saints where j == 2 for M = november
     type christmas where j == 25 for M = december
    end
 end
```

Figure 4: The Gregorian calendar in CaTTS-DL.

As with calendars, identifiers defined within a catalog are qualified, e.g. `ISO.Extended.stdfmt` is the full name of the above format. Date formats specified in CaTTS-FDL may be imported into a program in the language CaTTS-CL, XQuery, or any other language using calendric data typed after CaTTS-DL calendar specifications by CaTTS' import mechanism for formats `use_format`.

4.5 Use Case: Specifying Calendars

To express and find solutions to real-life, Web, Semantic Web, and Web Service related problems such as train scheduling or making appointments, context-dependent semantics considering both cultural and professional calendars and time zones is necessary.

Consider the following scenario: A Munich-based business man needs to schedule a phone conference with a colleague in Tel Aviv. Besides the various constraints to schedule the phone conference, the business men use different calendars. Both use the *Gregorian* calendar to schedule everyday times and dates, but shifted according to different *time zones.* The Tel Aviv-based business man uses the *Hebrew* calendar for holidays to be considered, but, for the same reason, the Munich-based business man uses the Gregorian.

The CaTTS-DL calendar signature in Figure 3, assigned to the identifier `STD`, describes standard calendar types and groups matched by most calendars. `second` is a non-further specified type identifier.

The calendar definition given in Figure 4 binds a calendar to the identifier `Gregorian` such that this calendar must match the calendar signature `STD` (denoted `Gregorian:STD`). CaTTS allows for importing external libraries using the reserved word `import.` The calendar `Gregorian` imports a library `LeapSeconds,` containing, among other things, a boolean function `hasLeapSec?` over minutes. Leap second insertion into UTC-time (recall that `reference` is the time granularity of UTC-seconds) started in 1972 (Gregorian); 10 leap seconds have been inserted into the first minute of the year 1972. In the present CaTTS-DL modeling, the index of this minute is directly referred to. The type identifier `second` is assigned to the predefined base type `reference.` The rules for the Gregorian leap month February are expressed by a suitable combination of operations predefined in CaTTS. The predefined arithmetic function `relative i in M` (used in the group `holiday`) selects specific data items `i` of type `day` relatively located to each data item in type `M,` a local macro variable substituted by a CaTTS-DL type. Any further type definition is straightforward following the rules of the Gregorian calendar [5].

The calendar definition given in Figure 5 binds a calendar to the identifier `Hebrew` such that this calendar must match the calendar signature `STD` (denoted `Hebrew:STD`). The Hebrew day "23 Tevet 5730", a Wednesday (yom revii) is the day corresponding to the Unix epoch ("1 January 1970" (Gregorian)). To implement an alignment of Hebrew regaim[4] and halaqim[5] (Hebrew partitions of hours) with CaTTS' `reference` type a respective shift (denoted by the definition of `_ref`) is defined. The type identifiers `second` and `minute` are used to match the Hebrew calendar with the standard signature `STD`[6]. Note that Hebrew weeks start on Sundays (yom rishon), and that the first month in any He-

[4]Plural of rega.

[5]Plural of heleq.

[6]Whether such an alignment of the Hebrew calendar to the

brew year is Tishri. Since Hebrew leap year computations depend on the Metonic cycle aligning 19 sun-based years to 235 lunar-based months, the index 1 for Hebrew months is respectively moved form "Tevet 5730" to "Nisan 5720" by resetting this month's index (133) relatively to the index 1 (denoted ~@133). To simplify the modeling of the Hebrew calendar in CaTTS-DL, the library `HebrewLeapYear`, containing the various functions (e.g. `isLeapAdarRishon?` and `isHebrewLeapYear?`) used in the present calendar specification is imported. Such functions may however be directly specified within a CaTTS-DL calendar definition by a user-defined "macro". E.g. `isLeapAdarRishon?` (i.e. the second to last month in a Hebrew year which depends on the Metonic cycle) can be specified by the following two CaTTS-DL macros:

```
macro isLeapAdarRishonInCycle?(i) =
  i ==  36 || i ==  73 || i ==  98 || i == 135
  || i == 172 || i == 209 || i == 234;
macro isLeapAdarRishon?(m) =
  isLeapAdarRishonInCycle?((m mod 235) + 1);
```

All rules implement in the CaTTS-DL definition of the Hebrew calendar are those suggested in [5].

Note that only a few Gregorian and Hebrew holidays are specified with the two present CaTTS-DL calendars. Further can be similarly specified in CaTTS-DL.

Since `reference` is UTC-time and since UTC-time is adjusted to the time zone Greenwich Mean Time (GMT), the previously mentioned calendars `Gregorian` and `Hebrew` correspond to this time zone. Any further calendar C matching also the calendar signature `STD`, but which refers to other time zones can be expressed by a (user-defined) calendar function, redefining the definition of type `day` by choosing suitable anchors for the considered time zones (cf. Figure 6). If the calendar function `EET` is applied to the calendar `Gregorian` or `Hebrew`, then any (aggregation or inclusion) subtype of `day` is respectively changed.

5. CATTS-CL: CONSTRAINT LANGUAGE

CaTTS-CL, CaTTS' constraint language, is statically typed after CaTTS-DL type definitions. CaTTS-CL is a language to declaratively express a wide range of temporal and calendric constraint problems. Such problems are then solved by CaTTS-CL's constraint solver. Given a CaTTS-DL specification of the Gregorian calendar (with types such as "weekend day") and the teaching calendar of a given university (with types such as "exam week"), one can refer in CaTTS-CL programs to weekend days. One can further express constraints on such days such as "after exam week" and "before June 2004" (assuming that the used date formats are specified in CaTTS-FDL). This problem is expressed in CaTTS-CL as follows:

```
X:weekend_day && X after Y:exam_week &&
X before "2004-06"
```
[7]

The constraint variable `X` ranges over values of type `weekend_day` and `Y` over `exam_week`. The constraints "after an examination week" (in CaTTS-CL `X after Y:exam_week`) and "before June 2004" (in CaTTS-CL `X before "2004-06"`)

`STD` calendar signature is appropriated or not, does not have to be discussed here. The present example aims at showing that it is possible and easy to express with CaTTS.

[7] In this and the following examples, constraint variables start with capital letters.

```
import HebrewLeapYear;
calendar Hebrew:STD =
 cal
   type _ref = refinement 114 @ reference(-43199);
   type second = aggregate 5 _ref;
   type rega = second;
   type minute = aggregate 76 second;
   type heleq = minute;
   type hour = aggregate 1080 minute;
   type day = aggregate 24 hour;
   type week = aggregate 7 day @ day(-2);
   type month = aggregate month(i)
     30 day named nisan,
     29 day named iyyar,
     30 day named sivan,
     29 day named tammuz,
     30 day named av,
     29 day named elul,
     30 day named tishri,
     alternate
      | newYearDelay(i) == 2
                  -> 30 day named long_marheshvan
      | otherwise -> 29 day named short_marheshvan
     end named marheshvan,
     alternate
      | newYearDelay(i) > 0
                  -> 30 day named long_kislev
      | otherwise -> 29 day named short_kislev
     end named kislev,
     29 day named tevet,
     30 day named shevat,
     alternate
      | isLeapAdarRishon?(i) -> 30 day
                                   named adar_rishon
      | otherwise none
     end,
     29 day named adar_sheni @ day(-21) ~@ 133;
   type adar = adar_rishon | adar-sheni;
   type year = aggregate
     alternate year(i)
      | isHebrewLeapYear?(i) -> 13 month
      | otherwise            -> 12 month
     end @ month(7) ~@ 5730;
   group day_of_week =
     with select day(i) where
      (relative i in week) == j
        type yom_rishon where j == 1
        type yom_sheni where j == 2
        type yom_shelishi where j == 3
        type yom_revii where j == 4
        type yom_hamishi where j == 5
        type yom_shishi where j == 6
        type yom_shabbat where j == 7
     end
   type weekend_day = yom_shishi | yom_shabbat;
   group holiday = with select day(i) where
    (relative i in M) == j
     type yom_kippur where j == 10 for M = tishri
     type passover where j == 15 for M = nisan
    end
 end
```

Figure 5: The Hebrew calendar in CaTTS-DL.

```
(*CET: Central European Time, GMT + 1, Munich*)
calendar_function CET(C:STD) : STD =
 cal
   type day = aggregate 24 C.hour @ C.hour(2);
 end
(*EET: Eastern European Time, GMT + 2, Tel Aviv*)
calendar_function EET(C:STD) : STD =
 cal
   type day = aggregate 24 C.hour @ C.hour(3);
 end
```

Figure 6: Calendars in different time zones in CaTTS-DL.

are straightforwardly expressed as illustrated. The constraints of CaTTS-CL are given in Appendix A.

CaTTS-CL provides function symbols for data items of types defined in CaTTS-DL, value constructors for time intervals and durations, arithmetic functions, and conditionals. CaTTS-CL's constraint symbols are common equation symbols, Allen's 13 interval relations [1], and the symbol ":" for type annotations. The complete syntax (of all CaTTS language fragments including CaTTS-CL) is given in Appendix A.

The constraint `X:`τ assigns to a variable `X` a finite domain over values of type τ (defined in CaTTS-DL). Note that the function symbols provided with CaTTS-CL are essentially the same as those provided with CaTTS-DL. Furthermore, the constraint symbols provided with CaTTS-CL have the same names as the predicate symbols provided with CaTTS-DL because type defining predicates and constraints are both boolean functions. However, there is an elementary difference between a CaTTS-CL constraint and a CaTTS-DL predicate: a constraint specifies the properties and relationships among partially unknown "objects". All possible solutions satisfying the constraint are computed by the constraint solver. In contrast, a predicate specifies the condition(s) to be satisfied by the elements belonging to some predicate subtype. E.g. the type defining predicate "holiday" specifies the infinite set of all holidays, and the constraint "X:holiday" computes the (infinite) set of all holidays. Thus, the values satisfying any CaTTS-CL constraint are computed, whereas a predicate only specifies the infinite set of values of some CaTTS-DL predicate subtype without performing any computations.

A CaTTS-CL *program* is a finite collection of CaTTS-CL constraints. Calendars defined in CaTTS-TDL are referred to by the `use_calendar` construct, data formats defined in CaTTS-FDL are referred to by the `use_format` construct, and external libraries are referred to by the `import` construct. The constraints specified in a CaTTS-CL program are delimited by the reserved words `prog` and `end`.

5.1 Use Case (Continued): Multi-Calendar Appointment Scheduling

Turning attention back to the two business men scheduling a phone conference (cf. Section 4.5) during the 1st week in October 2004, they specify the necessary constraints in a CaTTS-DL program (cf. Figure 7). The day, the phone conference might take place should be a Monday, Tuesday, Wednesday, or Thursday in Tel Aviv. The conference itself (denoted by the variable `Conf`) has a maximal duration of 1 hour, and it should be between 9 a.m. and 5 p.m. considering both time zones involved in this scenario. Furthermore, the telephone conference should take place in the first week of October 2004.

6. STATIC TYPE CHECKING AND CONSTRAINT REASONING SUMMARIZED

This section briefly summarizes the characteristics of CaTTS' language processors. A *static type checker* to ensure the behavior and semantics of calendric data and constraints, and a *constraint solver* to reason with such data.

6.1 Static Type Checker

In programming languages such as ML [9] static type checking is used as a "lightweight formal method" (i.e. merely syntactically tractable) for program analyses ensuring correct behavior of programs and/or systems w.r.t. some specification. CaTTS, however, uses static type checking for *semantic restrictions* of inherent ambiguous and/or imprecise calendric data and constraints ensuring correct interpretation of CaTTS-CL programs w.r.t. some CaTTS-DL calendar specification.

For space reasons, the typing and subtyping relations of CaTTS' static type checker cannot be presented in this article. First experimental results with a prototype implementation of CaTTS' type checker point to a good efficiency (i.e. type checking is polynomial). Further investigations concerning efficiency as well as soundness and completeness results are undergo.

6.2 Constraint Solver

CaTTS' constraint solver essentially works on arbitrary finite domains with type annotations after calendars defined in CaTTS-DL. It departs from constraint systems over finite domains [6] due to (i) typed constraint variables, and (ii) constraint variables that may range either over single values or over intervals of values; thus, reasoning over (possibly periodic) intervals. The constraint solver refers to and relies on the type predicates generated from a calendar definition in CaTTS-DL. This makes search space restrictions possible, and furthermore, obtains the semantics of calendric data and constraints introduced with CaTTS-DL type definitions.

7. RELATED WORK

CaTTS complements data type definition languages and data modeling and reasoning methods for the Semantic such as XML Schema [12], RDF [15], and OWL [14]: XML Schema provides a considerably large set of predefined time and date data types dedicated to the Gregorian calendar whereas CaTTS enables user-defined data types dedicated to any calendar. RDF and OWL are designed for *generic* Semantic Web applications. In contrast, CaTTS provides with methods *specific* to particular application domains, that of calendars and time.

CaTTS departs from time ontologies such as the KIF time ontology [8], the DAML time ontology [4], and time in OWL-S [10] in many aspects.

CaTTS considerably simplifies the modeling of specificities of cultural calendars (such as leap years, sun-based cycles like Gregorian years, or lunar-based cycles like Hebrew months) as well as the modeling of professional calendars often involving "gaps" in time (e.g. "working day"), "gapped" data items (e.g. data items of type "working week"), and periodic events (e.g. "CS123 lecture") due to predicate subtypes.

The well-known advantages of statically typed languages such as error detecting, language safety, efficiency, abstraction, and documentation whereas the two latter obtain particular interest due to overloaded semantics of calendric data apply to CaTTS, as well. Beyond this, CaTTS' static type checker provides both meta-type checking of predicate subtype definitions in CaTTS-DL and type checking of constraints in CaTTS-CL, obtaining the semantics of different time granularities even for reasoning with their granules.

CaTTS comes along with a constraint solver dedicated to calendar definitions in CaTTS-DL; this dedication makes considerable search space restrictions, hence gains in effi-

```
program TelConference =
 use_calendar unqualified Gregorian;
 use_calendar Hebrew, CET(C), EET(C);
 use_format unqualified ISO.Extended;

 prog
    Conf:hour during X:week && X during "2004-10" && relative index X in october == 1 &&
    relative index Conf in CET(Gregorian).day >= 9 &&
    relative index Conf in CET(Gregorian).day <= 17 &&
    relative index Conf in EET(Gregorian).day >= 9 &&
    relative index Conf in EET(Gregorian).day <= 17 &&
    Conf during EET(Gregorian).(monday|tuesday|wednesday|thursday)
 end
```

Figure 7: A scheduling problem in CaTTS-CL.

ciency, possible.

While (time) ontologies follow the (automated reasoning) approach of "axiomatic reasoning", CaTTS is based on a (specific) form of "theory reasoning", an approach well-known through paramodulation. Like paramodulation ensures efficient processing of equality in resolution theorem proving, CaTTS provides the user with convenient constructs for calendric types and efficient processing of data and constraints over those types.

CaTTS inherently differs from specification languages for events and temporal expressions in natural language text such as TimeML [7]. TimeML is a language for annotating temporal information in text corpora whereas CaTTS is designed as a statically typed language specialized in calendar and time modeling and reasoning, addressed to Semantic Web applications and Web Services.

8. CONCLUSIONS

This article has introduced CaTTS, consisting of

- CaTTS-DL, a definition language, itself consisting of
  - CaTTS-TDL, a type definition language and
  - CaTTS-FDL, a date format definition language
- CaTTS-CL, a constraint language typed by CaTTS-DL definitions.

CaTTS' predicate subtype constructors allow for defining arbitrary time granularities as types in CaTTS-DL. The two subtype relations aggregation subtype of and inclusion subtype of provide means for conversions between those types during constraint reasoning with calendric data of those types, particularly in CaTTS-CL. CaTTS' polytypic constructors provide a convenient and intuitive manner to specify periodic events. Interpreting calendars as types and their parameterization ensures maintenance and reuse of calendars, where the "type" of a calendar provides a summary of that calendar.

CaTTS facilitates the modeling and efficient processing of calendar and time data in Web and Semantic Web applications and Web Services, especially compared to ontology-based modeling and reasoning.

9. REFERENCES

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

[2] C. Bettini, S. Jajodia, and S. X. Wang. *Time Granularities in Databases, Data Mining, and Temporal Reasoning*. Springer-Verlag, 2000.

[3] F. Bry and S. Spranger. Towards a Multi-calendar Temporal Type System for (Semantic) Web Query Languages. In *Proc. 2nd Int. Workshop Principles and Practice in Semantic Web Reasoning*, LNCS 3208. Springer-Verlag, 2004.

[4] DARPA Agent Markup Language. *A DAML Ontology of Time*, 2002.

[5] N. Dershowitz and E. Reingold. *Calendrical Calculations: The Millennium Edition*. Cambridge University Press, 2001.

[6] T. Frühwirth and S. Abdennadher. *Constraint-Programmierung*. Springer-Verlag, 1997.

[7] B. Ingria and J. Pustejovsky. TimeML: A Formal Specification Language for Events and Temporal Expressions. 2004.

[8] Knowledge Systems Laboratories, Stanford. *Time Ontology in KIF*, 1994.

[9] R. Milner, M. Tofte, and R. W. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[10] F. Pan and J. R. Hobbs. Time in OWL-S. In *Semantic Web Services, AAAI Spring Symposium Series*, 2004.

[11] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[12] W3C, World Wide Web Consortium. *XML Schema Part 2: Datatypes*, 2001.

[13] W3C, World Wide Web Consortium. *Extensible Markup Language (XML) 1.0 (Third Edition)*, 2004.

[14] W3C, World Wide Web Consortium. *OWL Web Ontology Language Overview*, 2004.

[15] W3C, World Wide Web Consortium. *RDF Primer*, 2004.

APPENDIX

A. SYNTAX OF CATTS

A.1 Identifiers

t ∈ TyVar	*type identifiers*	long
c ∈ CalId	*calendar identifiers*	long
s ∈ CalSigId	*calendar signature identifiers*	
f ∈ CalFunId	*calendar function identifiers*	
p ∈ ProgId	*program identifiers*	

For each class of identifiers `X` marked "long" there is a class `longX` of *long identifiers*; if x ranges over `X` then *longx* ranges over `longX`. The syntax of the long identifiers is given by the following:

```
longx ::=                          long identifiers:
          x                        identifier
          c_1. ... .c_n.x          qualified identifier n ≥ 1
```

The long identifiers constitute a link between declarations and calendars.

A.2 Grammar

CaTTS grammar, including the syntactic forms for both language formalisms CaTTS-DL and CaTTS-CL is given in a BNF-like notation (⟨...⟩denote $n \leq 0$ repetitions).

```
e ::=                                      exprs:
        k                                  constant
        n ty                               duration, n ∈ ℕ
        binOp e e                          binary op
        unOp e                             unary op
        alternate x:ty ⟨| e → e⟩           alternate
        ce                                 constraints
ce ::=                                     constraints:
        x                                  variable
        x:ty                               domain constraint
        ce iRel ce                         interval relation
        ce aRel ce                         arithm. relation
        ce && ce                           conjunction

binOp ::=  shift forward | shift backward |
           relative to | + | − | relative in |
           ∗ | \ | mod | div |
           max | min
unOp ::=   duration_of | begin_of | end_of | index
iRel ::=   equals | before | after | starts |
           started_by | finishes | finished_by |
           during | contains | meets | met_by |
           overlaps | overlapped_by
aRel ::=   == | <= | < | > | >= | !=

ty ::=                                     types:
        longt                              type identifier
        reference                          reference
        refinement n @ e                   refine, n ∈ ℕ
        aggregate e⟨,e⟩@ e                 aggregation
        select x:ty where e⟨e⟩             inclusion
        |ty|                               duration
        ty & ty⟨& ty⟩                      conjunction
        ty | ty⟨| ty⟩                      disjunction
        ty \ ty                            except
        ty #< ty                           coarser-restrict
        ty #> ty                           finer-restrict
dcl ::=                                    decls:
        type t = ty                        type
        group t = wspec                    group
        dcl;dcl                            sequential
wspec ::=                                  with specs:
        with ty ⟨type t
         where e;e for t_1=t_2⟩

caldcl ::=                                 calendar decls:
        dcl                                declaration
        calendar calbind                   calendar
                                           empty
        caldcl;caldcl                      sequential
calbind ::=                                calendar binds:
        c⟨(:sige)⟩= cale
cale ::=                                   calendar exprs:
        cal caldcl end                     generative
        longc                              identifier
        f(cale)                            function application
```

```
fundcl ::=                                 fun decls:
        cal_fun funbind                    generative
                                           empty
        fundcl;fundcl                      sequential
funbind ::=                                fun binding:
        f(c:s):s' = cale

sigdcl ::=                                 s decls:
        calendar type sigbind              generative
                                           empty
        sigdcl;sigdcl                      sequential
sigbind ::=                                s bindings:
        s = sige
sige ::=                                   s exprs:
        sig spec end                       generative
        s                                  identifier
spec ::=                                   specs:
        type t <: ty                       aggregation
        type t c : ty                      inclusion
        group t c : ty                     group
        calendar c:sige                    calendar
        spec;spec                          sequential

progdcl ::=                                p decls:
        ce                                 constraint
        use_calendar⟨unqualified⟩
        longc_1 ...longc_n;                n ≥ 1
        import⟨unqualified⟩
        lib_1 ...lib_n;                    n ≥ 1
        use_format⟨unqualified⟩
        cate_1 ...cate_n;                  n ≥ 1
        program progbind                   generative
        empty
        progdcl;progdcl                    sequential
progbind ::=                               p binds:
        p proge
proge ::=                                  p exprs:
        prog progdcl end                   declaration
        p                                  identifier

catdcl ::=                                 cat decls:
        catalog catbind                    generative
        format fid:ty = d where e          format
                                           empty
        catdcl;catdcl                      sequential
catbind ::=                                cat binds:
        cd⟨:s⟩= cate
cate ::=                                   cat exprs:
cat catdcl end   generative
        cd                                 identifier
        cate.cd                            qual id
        catalog cd                         catalog
```