Calendars and Topologies as Types –

A Programming Language Approach to Modelling Mobile Applications

François Bry, Bernhard Lorenz, and Stephanie Spranger

Institute for Informatics, Univ. Munich, Germany
http://www.pms.ifi.lmu.de
contact: spranger@pms.ifi.lmu.de

Abstract. This article introduces a programming language approach to modelling spatio-temporal data using *calendars* and *topologies* specified as types. Calendric and topologic data appearing in Web applications are most often rather complex, sometimes involving different calendars and/or topologies. The basic principle is to model spatio-temporal data by means of *predicate subtyping*. This principle is used to define calendric and topologic data types representing granularities as well as conversions between those data types. A thesis underlying this work is that calendars and topologies are more conveniently expressed with dedicated language constructs and that calendar and topology data are more efficiently processed with dedicated reasoning methods than with general purpose "axiomatic reasoning" of e.g. ontology languages or theorem provers.

1 Introduction

This article introduces a programming language approach to modelling spatio-temporal data using *calendars* and *topologies* specified as types. Calendars are human abstractions of the physical flow of time. They enable to measure time in *time granularities* like day, week, working day, and teaching term. Topologies are human abstractions of (schematised) geographic objects and their relationships. Similarly to time granularities in a calendar, geographic objects in a topology may have different *location granularities* like city, district, and street. More abstractly, calendars and topologies both are finite collections of granularities related to each other, in general, in different manners.

The authors of the work reported about in this article claim that a programming language approach to calendars and topologies has similar advantages as types (res. objects) and type checking in functional (resp. object-oriented) programming. Types complement data with machine readable and processable semantics. Type checking is a very popular and well established "lightweight formal method" to ensure program and system behavior and to enforce high-level modularity properties. Types and type checking enhance efficiency and consistency of (modern) programming and modeling languages. Specific aspects of calendars

Acknowledgment: This research has been funded in part by the PhD Program Logics in Computer Science (GKLI) and 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).

and topologies make type checking with calendars and topologies an interesting challenge: a mobile application listing pharmacies in the surrounding of a (mobile) user will preferably only mention those that are currently open. Such an application refers to various topologic and calendric data. Types give such data their intended semantics, e.g. that some data refer to days. Static type checking ensures certain semantics on the data when processing them, e.g. to find an open pharmacy. The calendric data involved in such mobile applications are most often rather complex, sometimes involving different calendars (e.g. cultural calendars like the Gregorian and the Islamic and professional calendars) with various regulations and lots of irregularities (e.g. leap years). The topologic data involved often refer to different topologies like buildings or cities and to connections between such topologies. Furthermore, calendar data such as dates are probably more than any other data domain a subject to user interpretation: e.g. the date "12/02/2005" is interpreted in France as 12^{th} February 2005 while it is interpreted as 2^{nd} December 2005 in the US. Many traditional Web sites and pages refer explicitly or implicitly to such calendar data. In the current Web, such data can hardly be interpreted by computers. The vision of the Semantic Web is to enrich the current Web with well-defined meaning and to enable computers to meaningfully process such data.

This paper is devoted to a unifying view of calendars and topologies: time and location granularities are modelled as data types and are related in calendars and topologies, themselves specified as types. The basic principle of the programming language approach to modelling spatio-temporal data is *predicate subtyping* [1]. Predicate subtyping with predicate types is a stronger form of typing and subtyping enabling to encode more information in types, because the elements of a predicate type are described by a predicate set. Predicate sets are used to declaratively define (possibly infinite) sets. Predicate types have been widely investigated in type theory, logics, proof assistents, and theorem proving. The typing approach to calendars and topologies presented in this article uses predicate types in a different manner and not for theoretical, but instead practical purposes: predicate subtyping is used to define calendric and topologic data types representing granularities as well as conversions between those data types. A thesis underlying the work reported about in this article is that calendars and topologies are more conveniently expressed with dedicated language constructs and that calendar data and expressions are more efficiently processed with dedicated reasoning methods than with "axiomatic reasoning" of ontology languages like RDF and OWL.

2 Advantages of Types and Static Type Checking

Static type checking (i.e. verifying at compile time whether expressions and definitions in a program obey the typing rules of the language) is a very popular and well established "lightweight formal method" to ensure program and system behaviour and to enforce high-level modularity properties [1].

Types and static type checking is as useful and desirable with calendric data types and topologic data types as it is with whatever other data type: it catches

a significant number of errors before a program runs. Types are a valuable form of program documentation that rarely becomes outdated. They simplify locating definitions in libraries. Furthermore, typed languages gain in efficiency, because functions need not verified during run time. Types are backbones for module systems yielding in abstraction. Specific aspects of calendars and topologies make static type checking for such data an interesting challenge. Some basic problems not only of theoretical but also of practical importance concerning calendric and topologic data can be solved: (i) The problem of granularity conversion (discussed for calendric types in [2]), i.e. to cast the elements of one granularity (e.g. day) to those of another granularity (e.g. working day) is solved with the concept of *predicate types.* (ii) *Context-aware* modelling of calendars and topologies is possible, i.e. the type checker statically verifies according to which calendar/topology some data have to be interpreted. A type checking approach with calendric types is proposed in [3]. (iii) Constraint solving on calendric and topologic data is performed *independently, efficiently* and *without loss of semantics* of the data. The reason for this is that predicate types specify a conversion from one data type to another, coincidently obtaining some level of abstraction by means of granularities. Constraint solving on calendric data with different calendric types by means of conversion constraints is introduced in [2].

3 A unifying View of Calendars and Topologies

This section is a mathematical prolog that formally introduces the notion of granularity.

Definition 1. *An* **n-dimensional space** *is a pair* $(\mathcal{A}^n,<_{\mathcal{A}^n})$ *where* $\mathcal{A}^n$ *is an infinite set (isomorphic to* $\mathbb{R}^n$*) and* $<_{\mathcal{A}^n}$ *is a total order on* $\mathcal{A}^n$ *such that* $\mathcal{A}^n$ *is not bounded for* $<_{\mathcal{A}^n}$*. An element* $a=(a_1,...,a_n)$ *of* $\mathcal{A}^n$ *is called* **n-point**.

Note that since $\mathcal{A}$ is totally ordered (recall that it is isomorphic to $\mathbb{R}$), the total order is preserved over the Cartesian product of $\mathcal{A}\times\ldots\times\mathcal{A}$.

Definition 2. *Let* $(\mathcal{A}^n,<_{\mathcal{A}^n})$ *be an n-dimensional space.*
A **generalised subspace** s *over*$(\mathcal{A}^n,<_{\mathcal{A}^n})$ *is a (finite or infinite) collection of pairwise disjoint, totally ordered right-open subspaces* $[a^-,a^+)$, $a^-,a^+\in\mathcal{A}^n$ *such that* $a^-<_{\mathcal{A}^n}a^+$ *and* $a\in[a^-,a^+)$ *iff* $a\in\mathcal{A}^n$ *and* $a^-\leq_{\mathcal{A}^n}a<_{\mathcal{A}^n}a^+$.

In the following, $S_{\mathcal{A}^n}$ denotes the set of all generalised subspaces over $(\mathcal{A}^n,<_{\mathcal{A}^n})$.

Definition 3. *Let* $(\mathcal{A}^n,<_{\mathcal{A}^n})$ *be an n-dimensional space.*
Let $G=\{g_i \mid i\in\mathbb{Z}\}$ *be a set isomorphic to* $\mathbb{Z}$*. Let call the elements of* G **granules**. *A* **granularity** *is a (non-necessarily total) function* $\mathcal{G}$ *from* G *into* $S_{\mathcal{A}^n}$ *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* $a_i=(a_{i_1},...,a_{i_n})\in\mathcal{G}(g_i)$ *and for all* $a_j=(a_{j_1},...,a_{j_n})\in\mathcal{G}(g_j)$*, then* $a_i<_{\mathcal{A}^n}a_j$*, and*
2. *If* $\mathcal{G}(g_i)=\emptyset$*, then* $\mathcal{G}(g_j)=\emptyset$.

According to Definition 3, granules of a same granularity are totally ordered and non-overlapping. The first condition of Definition 3 induces from the ordering of the n-point (of the n-dimensional space) the common-sense ordering on granules. The second condition of Definition 3 is purely technical: it makes it possible to refer to the *infinite* set $\mathbb{Z}$ also for *finite* sets of granules.

Examples of granularities are *time granularities* [4] over $(\mathcal{A},<_{\mathcal{A}})$ *location granularities* over $(\mathcal{A}^2,<_{\mathcal{A}^2})$.

Granularities can be defined by specifying subtype relations (in terms of predicates). Two subtype relations, *aggregation of* and *inclusion of*, have been defined for (one-dimensional) time granularities in [4]. 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 of the type "day" since each week can be defined as a time interval of days.

Similar to the subtype relations, aggregation and inclusion, between time granularities, aggregations and inclusions are defined between location granularities as follows.

Definition 4. *Let $\mathcal{G}$ and $\mathcal{H}$ be location granularities.*
$\mathcal{G}$ is an **inclusion subtype of** *$\mathcal{H}$, denoted $\mathcal{G} \subseteq \mathcal{H}$, i.e. every granule of $\mathcal{G}$ is a granule of $\mathcal{H}$.*

For example, the location granularity "subway station" is an inclusion subtype of the location granularity "station", selecting only those stations with subway connection.

Definition 5. *Let $\mathcal{G}$ and $\mathcal{H}$ be location granularities.*
$\mathcal{G}$ is an **aggregation-enclosure subtype of** *$\mathcal{H}$, denoted $\mathcal{G} \preceq \mathcal{H}$, if every granule of $\mathcal{G}$ is a 2-dimensional space over $\mathcal{H}$ and every granule of $\mathcal{H}$ is included in (exactly) one granule of $\mathcal{G}$.*
$\mathcal{G}$ is an **aggregation-connection subtype of** *$\mathcal{H}$, denoted $\mathcal{G} \prec \mathcal{H}$, if every granule of $\mathcal{G}$ is a 2-dimensional space over $\mathcal{H}$ and every granule of $\mathcal{H}$ is included in (at least) one granule of $\mathcal{G}$.*

For example, the location granularity "train network" is an aggregation-enclosure subtype of the location granularity "train connection", aggregating a set of train connections into a network. And a "train connection" itself is an aggregation-connection subtype of "station", connecting several stations to a train line. Note that connections are not necessarily disjoint, because the same station may participate in different train lines, for example.

The two subtype relations, inclusion subtype and aggregation subtype, are corner stones of modeling granularities as types 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 and topologies. Indeed, they reflect widespread forms of common-sense reasoning with calendric and topologic data.

4 Modelling Calendars as Types in CaTTS

CaTTS is a generic modelling language [4] for data modelling and reasoning with calendars. 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. CaTTS-DL provides with CaTTS-TDL (for type definition language), a tool to define calendars and CaTTS-FDL (for format definition language), a tool to define calendar data, in particular dates to give calendar data well-defined meanings. CaTTS-CL provides a means to express a wide range of temporal constraints over calendar data referring to the types defined in calendar(s) specified in CaTTS-DL.

In CaTTS-DL, one can specify in a rather simple manner more or less complex, cultural and professional calendars. Irregularities like leap seconds or Hebrew leap months can be easily expressed in CaTTS-DL. In particular, CaTTS-DL provide a means to define time granularities as *calendric data types* by means of predicate types. E.g. the Gregorian calendar can be modelled in CaTTS-DL as follows:

```
calendar Gregorian =
 cal
   type second;
   type minute = aggregate 60 second @ second(1);
   ...
   type month = aggregate
     31 day named january,
     alternate month(i)
      | (i div 12) mod 4 == 0 &&
        ((i div 12) mod 100 != 0 || (i div 12) mod 400 == 0) -> 29 day
      | otherwise                                              -> 28 day
     end named february, ..., 31 day named december @ day(1);
   type year = aggregate 12 month @ month(1);
   type weekend_day = select day(i) where
            relative i in week == 6 || relative i in week == 7;
   type working_day = day\weekend_day;
 end
```

The above calendar specification binds a calendar (between the keywords `cal` and `end`) to the identifier `Gregorian`. This CaTTS-DL calendar specification consists of a set of type definitions (each identified by the keyword `type` followed by an identifier). The first type defined is `second`. It has no further properties. The type `minute` is defined from the type `second` by specifying a predicate. The CaTTS language processor interprets this recursive type definition as an *aggregation subtype* of the type `second` such that each of its elements comprises 60 seconds[1] (denoted `aggregate 60 second`) and that the minute that has index 1, i.e. `minute(1)` comprises all seconds between `second(1)` in `second(60)` (denoted `@ second(1)`). Any further type definition follows the same pattern. The definitions are straightforward following the rules of the Gregorian calendar [5]. Since Gregorian months have different lengths, a CaTTS type `month` is defined with a repeating pattern of the twelve months. The months February, which is one day longer in each Gregorian leap year is defined by an additional

[1] In CaTTS-DL, it is possible to define a type `minute` that considers leap seconds, as well.

pattern which specifies the leap year rule for the Gregorian calendar using the CaTTS language construct `alternate...end`. The type definition of the type `weekend_day` is derived from that of the type `day`. The CaTTS language processor interprets this type definition as an *inclusion subtype* of the type `day` such that each of its elements must be relatively to a week either the 6^{th} or the 7^{th} day (denoted `relative i in week == 6 || relative i in week == 7`). The type `working_day` is also and inclusion subtype of day, selecting those days which are not weekend days (denoted `day\weekend_day`).

The above exemplified CaTTS-DL calendar specification defines a calendar as a "type" that can be used, in principle, in *any* language (e.g. XQuery, XSLT, XML Schema), using calendar data enriched with type annotations after this CaTTS-DL calendar. CaTTS' type checker [3] is used to check the calendar data typed after a CaTTS-DL calendar in such programs or specifications, thus, providing a means to interpret such data.

Note further that particularities like time zones can be easily expressed in a CaTTS-DL calendar definition. Calendar definitions of other cultural calendars in CaTTS-DL, in particular the Islamic and Hebrew calendars and variations of the Gregorian calendar like the Japanese calendar as well as date format specifications using CaTTS-FDL are given in [4, 6, 7].

5 Topological Types

In many cases, location reasoning pertains to routing and navigation tasks which rely on network infrastructures. Here, we use networks as a straightforward example for a topology, knowing that more complicated topologies might require further research. However, this example shows that a holistic model of the real world is rarely necessary. E.g. a journey involving the public underground system can be planned without any information about the *geographic* composition of the subway network. Knowledge about schedules and the *topologic* structure suffices to find an optimal connection between two stations. Expressing such information in a location type language would provide both an abstraction from geographic coordinates and a means to enable reasoning on location data. Let us consider a part of Munich's subway and suburban train network defined in a topology type system.

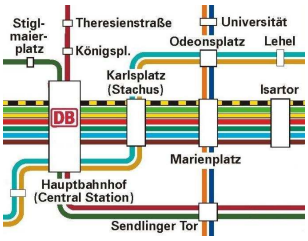

Fig. 1. Network section

```
type subway_station = collect
     Sendlinger_Tor, Hauptbahnhof, Marienplatz, Stiglmaierplatz, ...
type train_station = collect Hauptbahnhof, Karlsplatz, Marienplatz, ...
type station( subway_station | train_station );
type U1 = connect Sendlinger_Tor - Hauptbahnhof - Stiglmaierplatz - ...
type U3 = connect Sendlinger_Tor - Marienplatz - Odeonsplatz - ...
type S2 = connect Hauptbahnhof - Karlsplatz - Marienplatz - ...
type network = collect U1, U3, S2;
```

A network such as the one in figure 1 can be defined as shown above. The type `subway_station` is defined as a collection of named subway stations. A collection

is an ordered set of entities which satisfy certain constraints, in this case denoting a subway station. The definition of `train_station` is analogous. The common supertype `station` is a union of both (not necessarily disjoint) sets. A subway line (`U1`, `U3`, etc.) is defined as an ordered connection of subway stations. If there is a subway train servicing station A, B, and C (in this order), then these stations are connected with each other to form a line. Therefore, a line is directed, i.e. a service is usually comprised by two lines operating in different directions. This is especially useful for modelling different time tables and changing services for off-peak operation, etc. A network is in turn a collection of lines.

The approach skechted above provides a means for modeling topological data required in many applications pertaining to routing problems. Map representations are facilitated by linking the topological information to spatial data, e.g. sets of coordinates (polygons) which denote areas. This linking also facilitates the use of established calculi like RCC8 [8]. Furthermore, this approach allows for symbolic queries, such as finding out which stations lie in a certain quarter of the city or which district office is in charge of a road segment. Linked data is also necessary whenever for example polygons cannot be *directly* represented as location granularities, which is only possible in special cases (e.g. when areas can be regularly divided into same-size cells.)

6 Conclusion and Perspectives

This article has reported on a new approach introducing a unifying view of calendars and topologies in which time and location granularities are modelled as data types. Further work will be devoted to the refinement the idea of bridging between temporal and spatial modelling and reasoning techniques and their application to typical problems in common scenarii.

References

1. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
2. Bry, F., Rieß, F.A., Spranger, S.: A Type Language for Calendars. submitted to publication (2005)
3. Bry, F., Rieß, F.A., Spranger, S.: A Reasoner for Calendric and Temporal Data. submitted to publication (2005)
4. Bry, F., Rieß, F.A., Spranger, S.: CaTTS: Calendar Types and Constraints for Web Applications. In: Proc. 14^{th} Int. World Wide Web Conference, Japan. (2005)
5. Dershowitz, N., Reingold, E.: Calendrical Calculations: The Millennium Edition. Cambridge University Press (2001)
6. Bry, F., Spranger, S.: Towards a Multi-calendar Temporal Type System for (Semantic) Web Query Languages. In: Proc. 2^{nd} Int. Workshop Principles and Practice in Semantic Web Reasoning. LNCS 3208, Springer-Verlag (2004)
7. Bry, F., Haußer, J., Rieß, F.A., Spranger, S.: Cultural Calendars for Programming and Querying. In: Proc. 1^{st} Forum on the Promotion of European and Japanese Culture and Traditions in Cyber Society and Virtual Reality, France. (2005)
8. Randell, D.A., Cui, Z., Cohn, A.: A Spatial Logic Based on Regions and Connection. In: Proc. 3^{rd} Int. Conf. Principles of Knowledge Representation and Reasoning, USA. (1992) 165–176