Frank, A. U., and A. Gruenbacher. "Temporal Data: 2nd Order Concepts Lead to an Algebra for Spatio-Temporal Objects." Paper presented at the Workshop on Complex Reasoning on Geographical Data, Cyprus (1 December 2001) 2001.

Temporal Data: 2nd Order Concepts lead to an Algebra for Spatio-Temporal Objects

Andrew U. Frank and Andreas Grünbacher

Institute for Geoinformation, Technical University of Vienna
{frank, gruenbacher}@geoinfo.tuwien.ac.at

Abstract. The need for spatio-temporal data is widespread and obvious, but current commercial GIS have only very limited support for it. The difficulties with temporal data seem to be connected to the preference for first order formalisms underlying those systems. This paper proposes a second order method to describe objects that change in time, and describes an algebra over such objects.
The paper concludes with some general comments about the focus of computer science on algorithms and their performance, and the need for distinction from information science, with its focus on the semantics of data and operations.

1 Introduction

The world is ever changing. Some objects change so slowly that we have the illusion of a static situation. This is the view of the world used in topographic mapping, where only objects that change very slowly are included. Geographic information systems that are influenced by the traditions of cartography tend to show a static situation of the world, which we call a *snapshot.* We all know that maps are quickly out of date due to the changes in the world. Considerable resources are spent to keep geographic information systems used for administrative purposes up to date – but still, these representations remain static snapshots of reality.

Administrative databases were designed to maintain the current (valid) state of knowledge about the world in an organization, and make the data available to every process that needs them. Database theory is related to first order predicate calculus; indeed there is a close connection between relational database theory (Codd, 1970, 1982; Ullman, 1988) and relation calculus (de Moor, 1992; Schröder, 1966) (Gallaire et al., 1984). The theory for changing properties of a situation is situation calculus (McCarthy and Hayes, 1969), which is best used in the form recently given by Reiter (to appear). The use of temporal logic seems to bring unnecessary complications.

Administrative applications in the public administration require the history of cases, and must be able to trace all changes to demonstrate the legality of the procedures. Similar requirements follow from accounting and audit rules. Some form of history management is possible without full-fledged support for temporal data (Snodgrass, 1992).

New requirements are posed by systems to trace moving cars or airplanes in flight; such examples motivate the approach suggested here. Managing data of changing spatial objects, where boundaries or other attributes change in time, seems to be even more difficult (Frank et al., 2000).

Generally, research on the treatment of spatio-temporal data advances only slowly. The topic was included in the original NCGIA research agenda (NCGIA, 1989) and subject to an initial workshop at the University of Maine (Barrera et al., 1991). The Ph.D. theses of Langran (1992) and Al-Taha (1992) dealt with two different limited aspects, namely land use and cadastre, but only in 1993 a specialist

meeting of the NCGIA approached the issue (NCGIA, 1993). The number of contributions about time related topics in the series of conferences on spatial information theory which listed temporal aspects among the topics of interest increased very slowly from 1992 to 2001 (COSIT'2001 conference). The Chorochronos project[1] advanced this fundamental theme further, but we are still far from a complete theory and a corresponding practice.

This contribution suggests that the difficulties with temporal data can be linked to the extensive usage of first order languages in computer science. Simply put, first order languages are languages where variable symbols stand for individuals or properties of individuals. Second order languages allow variable symbols to stand for functions as well. Second order formalisms will be used in the remainder of this paper to show a compact formalization for spatio-temporal objects with a corresponding algebra.

The hypothesis of this paper is that all operations defined for a static situation carry over almost trivially to time synchronous operations on changing values. This paper is limited to the case of synchronous operations on changing dependent objects, but the extension to operations that cover more than a single point of time seems possible.

We have experimented with an implementation of the approach presented in this paper using the functional programming language Haskell (Peterson et al., 1996). Haskell is a language with multiple parameter types in which type checking of second order operations is possible. The examples that are included in this paper are essentially valid Haskell 98 code (Peyton Jones et al., 1999) with standard extensions, namely overlapping instances and multiple parameter classes. To allow an easier understanding some minor details that are not essential for the discussion are omitted. The full executable source code is available from our Web site, `http://www.geoinfo.tuwien.ac.at/`. For those readers who are not familiar with functional programming or Haskell, Hudak et al. (1997) offer a gentle introduction into these topics.

2 Moving Objects

The representation of moving objects is important for applications which keep track of cars, aircrafts or similar objects, which change their position or attributes quickly. We can model the position and movement of an object in space as a time varying vector. With no loss of generality we limit our examples to two dimensions in space.

In the presented model the positions of objects are time continuous functions. In real-world applications, often only the positions of objects at discrete instants of time are known (i.e., the positions are sampled at constant or varying time intervals). Assuming continuous movements of the objects in the real world, linear or higher-order interpolation can be used to reconstruct plausible intermediate positions.

The following pieces of Haskell code show how a model of moving points can be built. We assume that the language provides a suitable representation of floating point numbers, and the usual set of operations on them. Based on that we define the class *Vectors* and operations that can be carried out on vectors. A vector can be constructed from two Cartesian coordinates, and each of the coordinates of a given vector can be determined.

```
class Vectors v c where
  xy :: c → c → v c
  x, y :: v c → c
```

[1] Chorochronos project, `http://www.dbnet.ece.ntua.gr/~choros/`

In the examples in this paper we represent vectors by their Cartesian coordinates. Different representations, like polar coordinates, could be used, but we do not want to do this here. We also define how the base vector operations apply to this representation.

```
data Vector c = Vector c c -- one possible representation
instance Vectors Vector c where
   x (Vector x1 y1) = x1
   y (Vector x1 y1) = y1
   xy x1 y1 = Vector x1 y1
```

We want to define the "+" and "−" operations for vector addition and subtraction. Since all operations that are defined for multiple parameter types are defined in *classes* in Haskell, and the "+" and "−" operations can be defined for arbitrary numbers, we make our vectors an instance of the class *Numbers.* We allow the coordinates of a vector to be numbers of any type (that is what the context "`Number c` ⇒" stands for), and define how the vector operations are implemented, using the operations defined for numbers.

```
instance Number c ⇒ Number (Vector c) where
   a + b = xy (x a + x b) (y a + y b)
   a - b = xy (x a - x b) (y a - y b)
```

The coordinates of a vector can be of various types. Static vectors like *v1* as well as time varying vectors like *v2* and *v3* can be defined. We define two new types *Time* and *Changing* for representing time, and for representing objects that change over time. *Changing* is equivalent to Güting's τ operator, which is defined as $\tau(\alpha) = \underline{time} \rightarrow \alpha$ in Güting et al. (2000). Just as the type of the coordinates is a parameter of type *Vector* in the previous example, the type of the object that is changing is a parameter of type *Changing.*

```
v1 :: Vector Float
v1 = xy 1.0 0.0 -- a static vector

type Time = Float
type Changing t = Time → t

v2, v3 :: Changing (Vector Float)
v2 t = xy (2.0 * t) (3.0 * t)

-- v2 2.0 == xy 4.0 6.0
```

It is possible to define changing vectors like *v2,* just as well as it is possible to define vectors of changing coordinates like *v4.* For defining *v4* we define two helper functions that determine each of the coordinates of *v2* as functions of time.

```
v2_x, v2_y :: Changing Float
v2_x t = x (v2 t)
v2_y t = y (v2 t)

v4, v5 :: Vector (Changing Float)
v4 = xy v2_x v2_y
```

The definition of *v5* shows a more compact way of defining a vector of changing coordinates, using so-called *lambda expressions.* Lambda expressions are nothing more than functions without names. The variables between λ and the arrow define the (bound) variables; the expression after the arrow defines the value of the function. The function *eval* takes a vector of changing coordinates, and converts it into a changing vector. The sub-expression `x v t` in the definition of *eval* has the following meaning: Take the x coordinate of the vector v, and evaluate this function at time t. Since v is a vector of changing coordinates, its x coordinate is a changing value.

A changing value is defined as a function from time to a value, as the definition of *Changing* shows. When some time is passed to this function, the result is a vector.

```
v5 = xy (λ t → 1.0 - 100.0 * t) (λ t → 1.5 - 150.0 * t)

eval :: Vector (Changing Float) → Changing (Vector Float)
eval v t = xy (x v t) (y v t)

v3 = eval v5
```

The step of converting a changing vector to a vector of changing coordinates can be expressed abstractly. Here the sub-expression `λ t → x (a t)` has the following meaning: When given a time value, evaluate the changing vector a at this time t, and return the x coordinate of the result.

```
conv :: Changing (Vector Float) → Vector (Changing Float)
conv a = xy (λ t → x (a t)) (λ t → y (a t))

-- two equivalent definitions:
v4 = conv v2
v2 = eval v4
```

Obviously *conv* is the inverse operation of *eval.* In the language of category theory, $conv \circ eval = eval \circ conv = id$ (the symbol $\circ$ stands for functional composition; id is the identity function).

There is only a small conceptual difference between changing vectors and vectors of changing coordinates. We will later see that vectors of changing coordinates are sometimes preferable over changing vectors, and sometimes that the opposite is the case. For example, vectors of changing coordinates allow operations on static coordinates to be carried over to operations on changing coordinates.

3 Algebra over moving points

In second order languages, functions are "first class citizens," and can be used as arguments to other functions. Therefore, changing objects that are conceptualized as functions can be used as arguments to other operations. As an example let *v2* be a passenger, and *v3* a train this passenger is in. The position of the passenger as seen from the outside (provided that the train and the passenger are both slow enough that relativistic effects are not significant) can be computed by adding *v2* and *v3.* The following example defines the function *plus,* which adds two such changing vectors (of floating point coordinates). We will later see that this operation can also be defined more elegantly. The type *ChgVecFloat* simply serves as an abbreviation for its more verbose definition. The "+" operation used is the one that has just been defined.

```
type ChgVecFloat = Changing (Vector Float)
plus :: ChgVecFloat → ChgVecFloat → ChgVecFloat
plus a b t = (a t) + (b t)

v6 :: Changing (Vector Float)
v6 = v2 `plus` v3
```

Semantically the *plus* operation is a *synchronous* combination of values, i.e., both objects *v2* and *v3* are observed at the same instant of time, and the result also applies to this instant of time only. We do not consider asynchronous combinations of operations, like the minimal distance between two moving objects at independent times, or effects on semantically linked time-dependent values, like the position and the speed vector of an object, in this paper.

4 Generalization: Changing Values

The approach described so far is not limited to modeling changing locations of objects. It is also possible to model arbitrary changing attributes of objects using this approach. Moving objects are just one specific instance of changing objects. Properties of objects that do not change over time are a special case in that the values of these functions stay constant no matter what value their time parameter attains.

An interesting operation on moving objects is the distance between two moving objects. Assuming there is an operation *dist* that computes the distance between two vectors, the distance between moving vectors can be computed by the *dist'* operation. Note the similarity of the *plus* and the *dist'* operations. As for the *plus* operation, a formalism exists with which we can derive *dist'* from *dist* more elegantly (we introduce this formalism in the following section).

```
dist' :: ChgVecFloat → ChgVecFloat → Changing Float
dist' a b t = dist (a t) (b t)

d23 :: Changing Float
d23 = dist' v2 v3
```

5 Lifting: Operations carry over from non-changing values

Many operations with time dependent parameters have synchronous semantics, and are defined as the time synchronous application of the corresponding static operations (i.e., the values for computing the operation are all taken at a single instant of time, similar to a photographic snapshot). The distance between moving points, presented before, is one such example.

A range of functions is usually defined for manipulating values in snapshots, like addition, subtraction, multiplication, etc. Functions in the corresponding systems that model temporal aspects are time synchronous. It is possible to transform such functions into a temporal algebra; this transformation is usually called *lifting.*

We have already seen how the operations "+" and *dist* on vectors are lifted into the temporal domain; this resulted in the *plus* and *dist'* operations. Lifting can be defined abstractly.

In the following examples we show how operations for lifting functions from the static into the temporal domain can be defined. The lift operations for constant values, for functions with one, two and three parameters, have the following type signatures. We put this set of operations in the class *Lifts* so that we can use the same operations for lifting operations into arbitrary domains, and are not limited to the temporal one. This is the same step of abstraction done for operations like "+", which can be implemented for natural numbers, real numbers, complex numbers, vectors, changing vectors, etc.

```
class Lifts a where
  lift0 :: a → f a
  lift1 :: (a → b) → f a → f b
  lift2 :: (a → b → c) → f a → f b → f c
  lift3 :: (a → b → c → d ) → f a → f b → f c → f d
```

It is easy to see that the *lift2* operation, when applied to the *dist* operation, results in the *dist'* operation if f is substituted by the type *Changing:* The resulting function then takes two parameters of type *Changing (Vector Float),* and returns a result of the same type. An implementation of the lift operations for the temporal domain is given below. With this definition, we can give a simpler definition of *dist',* which is also shown.

```
instance Lifts Changing where
   lift0 a = λ t → a
   lift1 op a = λ t → op (a t)
   lift2 op a b = λ t → op (a t) (b t)
   lift3 op a b c = λ t → op (a t) (b t) (c t)

-- a simpler definition:
dist' = lift2 dist
```

Arbitrary operations can now be lifted into the domain of changing values. The operations on changing numbers (like the distance between two moving objects) are one example. We define the operations "+", "−", "∗" and *sqrt* (the square root) for changing numbers by lifting the existing operations on static numbers into this domain. Then we define three changing numbers. The third changing number uses the "+" operation defined here.

```
instance Number n ⇒ Number (Changing v) where
   (+) = lift2 (+)
   (-) = lift2 (-)
   (*) = lift2 (*)
   sqrt = lift1 sqrt

f1, f2, f3 :: Changing Float
f1 = sin -- the standard sine function
f2 = lift0 1 -- a changing constant
f3 = f1 + f2
```

Since we have declared vectors to be numbers (see the definition of the "+" and "−" operations on page 3), the above definition of operations for changing numbers automatically also results in operations on vectors of changing coordinates being defined.

In Haskell type definitions can have parameters, and that these type parameters are eventually filled in with concrete types (such as *Float*, *Time* → *Float*, etc.). For a *Vector,* the only parameter is the type of the coordinate values used. We can make all operations defined on vectors available for a new type by making the new type suitable as the coordinate parameter, and by defining the *lift* operations for this parameter.

Logical operations (e.g., an operation that determines whether an object is within a given distance from another object) can be lifted under the same preconditions. Unfortunately the definitions provided for logical operations in the Haskell Prelude, a library of standard definitions that is available in all Haskell programs unless programs explicitly request that it should not be included, do not allow this flexibility. While for all of these operators the type of the objects to be compared is a parameter, the type of the result of these operations is fixed.

```
data Bool = False | True
class Ord a where
   (<), (≤) :: a → a → Bool
```

Fortunately it is possible to provide a more flexible definition of these operators that also allows these operators to be lifted, by also making the result of these operations a parameter. We give an example of lifting the comparison operations, and start with a more flexible definition of the operators themselves.

```
class Ord a bool where
   (<), (≤) :: a → a → bool
```

All the comparison operations between the built-in types, which are defined in the Haskell Prelude, can easily be made available. The following piece of code shows this step.

```
class Prelude.Ord a ⇒ Ord a where
   (<) = (Prelude.<)
   (≤) = (Prelude.≤)
```

With these definitions we can define an operation that determines whether an object is within a specified range of another object in the static domain. We can then lift this operation into the domain of changing values. Note that in this example we are lifting vectors into changing vectors, unlike before, where we have lifted vectors into vectors of changing coordinates.

```
isClose :: Vector Float → Vector Float → Bool
isClose a b = dist a b < 1.0

b1 :: Bool
b1 = isClose v1 (xy 0.0 0.0)

b2, b3 :: Changing Bool
b2 = (lift2 isClose) v2 v3

-- b2 0.0 == True
-- b2 2.0 == False
```

It is of course also possible to create a similar test function for vectors of changing coordinates using *eval,* the conversion function from vectors of changing coordinates to changing vectors.

```
isClose' :: Vector (Changing Float) → Vector (Changing Float) → Changing Bool
isClose' a b = (lift2 isClose) (eval a) (eval b)

b3 = isClose' v4 v5
```

6 Changing objects in a changing world

The observation made in the previous section points to an important difference in conceptualization: we can perceive the world as a changing value, from which we can deduce a particular value, a snapshot, for a specific instant in time. We can also model the world as a collection of time dependent objects. The objects themselves are then time dependent functions. The following code models this hierarchy of types. (In this example a qualifier identifies a specific object, or an attribute of an object, similar to a name.)

```
-- The world can be modeled as one of these types:
type World = Time → Snapshot
type World = Identifier → ChangingObject

type Object = AttributeName → Attribute
type Snapshot = Identifier → Object
```

The same ambiguity in conceptualization occurs for changing objects, which can be seen as objects changing in time, or as a set of changing attributes. (This hierarchy may not even stop at the attribute level; the attributes may also be composed of changing values, etc.).

```
-- A changing object can be modeled as one of these types:
type ChangingObject = Time → Object
type ChangingObject = AttributeName → ChangingAttribute

type ChangingAttribute = Time → Attribute
```

These different views are alternative ways of conceptualizing the world, and they are also alternatives in an implementation.

7 Semantics vs. Efficiency

This paper has approached the problem of temporal data from a purely conceptual point of view, using advanced mathematical formalisms, namely algebra and category theory. This has allowed us to carry over the operations for values in a snapshot to time changing values. It was possible to formalize this extension in a general way, such that all operations (e.g., the calculation of a distance) once defined for single values, can also be applied to changing values. In this approach details of the efficiency of an implementation are hidden behind abstractions, and do not blur the picture when specifying the semantics of operations.

One may see a difference in concerns between a Computer Science and an Information Science approach. Computer science in the tradition of Donald Knuth (Knuth, 1973) studies the design of algorithms and data structures, and optimizes them for efficiency. The big-O notation is a fundamental tool to describe the asymptotical performance of algorithms (i.e., how the performance of algorithms changes depending on the size of the problem), and to compare different implementations that are semantically equivalent.

Information science should only be concerned with the specification of the semantics. For an Information Scientist it is sufficient to have a trivial implementation that computes the desired results, with no concern for the performance of the implementation. Once the semantics are understood well enough, it is possible to look for more efficient algorithms.

A division of labor between information science and computer science allows each field to concentrate on what it is best at.

When developing computer software, people often try to optimize the solution for a problem before the problem is sufficiently well understood, which finally leads to more complex solutions. As computer scientists are developing increasingly powerful solutions for transforming simple implementations with poor performance into fast implementations (Bird and de Moor, 1997), such optimizations are increasingly becoming dispensable.

8 Conclusions and Future Work

If we consider the changing objects and changing values in this ever-changing world as functions, the ordinary calculations based on snapshots carry over to changing values. The formalism presented leads to the extension of snapshot operations (for single instants of time) to synchronous calculations for time series. The implementation of the proposed algebra for time dependent objects and values is not constrained. Several representations close to what good programmers would come up with are possible. This paper argues that the separation of the conceptualization from implementation aspects leads to better and more general solutions.

An alternative approach for dealing with changing values that uses intervals at the algebraic interface level (Güting et al., 2000) is more complex, and therefore more difficult to understand. The implementation of what we propose here is essentially the same, but most complications are hidden from the user.

The discussion in this paper was constrained to time synchronous operations. There certainly exist operations with complex temporal interrelationships, like the relationship between the position and the speed of an object (the one is a derivative of the other). We leave these aspects for future work.

Acknowledgments

This idea emerged from the cooperation in the Chorochronos project, and in particular from discussions with Ralf Güting. The first author is very grateful to Mark

Jones, who introduced him to Gofer and Haskell – and thus to a functional programming language with many high-level concepts built in – and pushed for classes with multiple type parameters, which are also used in the examples presented. A discussion with Stella Frank about the optimization of algorithms was also helpful in writing this paper.

This work was supported by the the REVIGIS project, which is being funded by the European Commission.

Bibliography

Al-Taha, K., 1992. Temporal Reasoning in Cadastral Systems. Ph.D. thesis, University of Maine.

Barrera, R., Frank, A., Al-Taha, K., 1991. Temporal Relations in Geographic Information Systems: A Workshop at the University of Maine. SIGMOD Record 20 (3), 85–91.

Bird, R., de Moor, O., 1997. Algebra of Programming. Prentice Hall International Series in Computer Science. Prentice Hall Europe, London.

Codd, E., 1970. A Relational Model of Data for Large Shared Data Banks. Communications of the ACM 13 (6), 377 – 387.

Codd, E., 1982. Relational data base: A practical foundation for productivity. Communications of the ACM 25 (2), 109–117.

de Moor, O., 1992. Categories, relations and dynamic programming. Ph.d. thesis, Oxford University.

Frank, A. U., Raper, J., Cheylan, J.-P. (Eds.), 2000. Life and Motion of Socio-Economic Units. GISDATA Series. Taylor & Francis, London.

Gallaire, H., Minker, J., Nicolas, J.-M., 1984. Logic and Databases: a deductive approach. ACM 16 (2), 153–184.

Güting, R., Böhlen, M., Erwig, M., Jensen, C., Lorentzos, N., Schneider, M., Vazirgiannis, M., 2000. A Foundation for Representing and Querying Moving Objects. ACM Transactions on Database Systems 25 (1), 1–42.

Hudak, P., Peterson, J., Fasel, J. H., March 1997. A Gentle Introduction to Haskell. Tutorial, Yale University.

Knuth, D., 1973. The Art of Computer Programming. Addison-Wesley, Reading, Mass.

Langran, G. (Ed.), 1992. Time in Geographic Information Systems. Technical Issues in GIS. Taylor and Francis.

McCarthy, J., Hayes, P. J., 1969. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In: Meltzer, B., Michie, D. (Eds.), Machine Intelligence 4. Edinburgh University Press, Edinburgh, pp. 463–502.

NCGIA, 1989. The Research Plan of the National Center for Geographic Information and Analysis. International Journal of Geographical Information Systems 3 (2), 117 – 136.

NCGIA, May 8-11, 1993 1993. Time in Geographic Space, I-10 Specialist Meeting. Tech. rep., NCGIA.

Peterson, J., Hammond, K., Augustsson, L., Boutel, B., Burton, W., Fasel, J., Gordon, A. D., Hughes, J., Hudak, P., Johnsson, T., Jones, M., Peyton Jones, S., Alastair, R., Wadler, P., 1996. Report on the functional programming language Haskell, Version 1.3.

Peyton Jones, S., Hughes, J., Augustsson, L., 1999. Haskell 98: A Non-strict, Purely Functional Language.

Reiter, R., to appear. On knowledge-based programming with sensing in the situation calculus. ACM Transactions on the Computational Logic .

Schröder, E., 1966. Algebra der Logik, I-III, reprint.

Snodgrass, R. T., 1992. Temporal Databases. In: Frank, A., Campari, I., Formentini, U. (Eds.), Theories and Methods of Spatio-Temporal Reasoning in Geographic Space (Int. Conference GIS - From Space to Territory, Pisa, Italy). Vol. 639 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, pp. 22–64.

Ullman, J. D., 1988. Principles of Database and Knowledgebase Systems. Vol. 1 of Principles of Computer Science Series. Computer Science Press, Rockville, MD.