Modelling Databases with Functional Programming

Damir Medak
Dept. of Geoinformation
Technical University Vienna
dmedak@geoinfo.tuwien.ac.at

1. INTRODUCTION

Recent developments in GIS show the lack of tools for managing temporal information, although the related research has been intensified during last 10 years. As the needs for spatio-temporal databases grew, different applications sought different interpretation of temporal component. A plethora of different temporal data models and query languages shows that a compact core of simple temporal operations is needed.

This paper discusses one particular type of change the objects in the real world could be exposed to: the change of objects' identity. Depending how they perform such essential change, all objects may be grouped in a few categories. These groupings are called lifestyles. A compact framework for dealing with temporal databases is given as an executable specification written in functional programming language. It is shown that apparently different domains share the same lifestyle. The example of cadastre parcels behaving like liquid objects emphasizes the point.

2. FRAMEWORK FOR LIFESTYLES

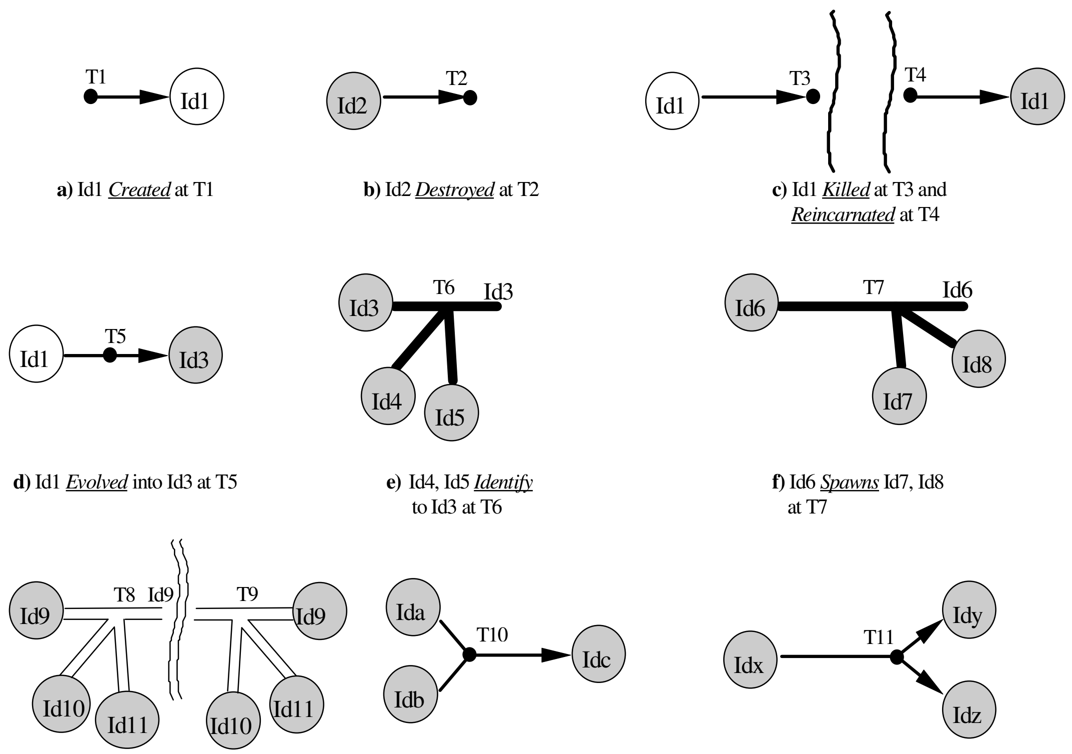
Before we start to analyze the temporal processes, it is necessary to informally describe how the model of the real world is constructed. The elements of the world are *objects* of different kinds which all have just one property in common: they have *identifiers* unique within the *universe of discourse*. This feature enables the system to distinguish one object from another. The identifier already used should not be given again even if the former object has been destroyed. The change in identifiers of the is studied in detail because it captures very low level of universality in the behavior of all objects. The most primitive operations for handling this type of change are called *temporal constructs*.

TEMPORAL CONSTRUCTS

The question when and how the objects change their identities is very important for the temporal databases. The work of many authors was summarized in (Al-Taha and Barrera 1994) and the list of applicable temporal constructs was proposed. These are shown on the figure 1.

The first two temporal constructs seem naturally connected: the object is created and some time later destroyed from the system. While the act of creation is indisputable, the concept of *destroying* needs short explanation. Namely, the goal of a temporal database is to keep the track about past times. Whatever was created at some point in time, must be observable at that particular time. Therefore, *destroy* should be understand as disabling the object from some point in history onwards. The information about its former existence is kept and can be easily retrieved (Clifford and Croker 1988).

On the other hand, *kill/reincarnate* construct represent the change of identity to inactive allowing future re-activating. This is not applicable to the biological life of human beings, but can be seen in their professional career: if one changes the job for a while and then comes to the old enterprise again, it would be a reincarnation, at least from the point of view of the employer.



a) Id1 *Created* at T1

b) Id2 *Destroyed* at T2

c) Id1 *Killed* at T3 and *Reincarnated* at T4

d) Id1 *Evolved* into Id3 at T5

e) Id4, Id5 *Identify* to Id3 at T6

f) Id6 *Spawns* Id7, Id8 at T7

g) Id10, Id11 *Aggregate to* Id9 at T8 Id9 *Disaggregates* at T9

h) Ida, Idb *Fuse into* Idc at T10

i) Idx *Fissions into* Idy, Idz at T11.

Figure 1: Temporal constructs of identities, (Al Taha and Barrera 1994)

Evolving can be seen as a combination of a *create* and *destroy*: one object leaves and another comes. But, the new object keeps the information about its ancestor. The concept of ancestoring is very helpful for tracking the history as it underlies all other temporal constructs with exception of aggregates.

The primary role of *identify* was to give the same identity to several objects coming from different sources and referring to the same object. However, it is useful for solving disputes which often emerge in land information systems. It is quite common practice in cadastre offices that bigger parcel retain its identifier after unification with "smaller part". Such an action was justified in savings of new numbers and related manual work in paperback cadastre. The same example applies to the temporal construct *spawn*. An object produces next object(s) continuing to have the same identifier.

The concept of *aggregation* relies on part/whole relation, which is both well explored and intuitive. The relation is hierarchical; parts may have parts. Standard example is a car having its chassis number (identifier), engine (which have its own number), steering wheel, tires etc. The aggregation is natural approach when dealing with solid objects, especially with manufactured goods.

While *an aggregation* is a reversible operation, *a fusion* (merging) is not, because the objects involved cease to exist. On the other hand, *a fission* (splitting) may be reversed if whole process can be reconstructed. However, the rules which govern the right of reincarnation are complex and shall not be discussed further. The concept of *fission/fusion* captures the behavior of liquids, which are more complicate stuff than solids, as shown in (Hayes 1985).

3. FORMALIZATION OF LIFESTYLES

The functional programming environment GOFER (Jones 1994) is used for formalization. The formalization starts with the signatures for the identity class. Two operations are defined for the class, *nextID* derives following identity for the object. The code is completely independent of implementation, i.e. how an ID is represented in the system. It is important that every representation conforms with predefined equality class, the context of IDs class. More about class based functional programming can be found in (Frank et al. 1997).

```
class Eq i => IDs i where
      nextID :: i -> i
```

Then, a class *Objects* with fundamental operations on objects is defined:

```
class Objects o where
      putID :: ID -> o -> o
      getID :: o -> ID
      kill, reincarnate :: o -> o
```

The class *Worlds* "wraps" all objects together, inheriting all operations from class *Objects*:

```
class Objects t => Worlds w t where
      createObj  :: t -> w t -> w t
      getObj     :: ID -> w t -> t
      destroyObj :: ID -> w t -> w t
      perform    :: (t -> t) -> ID -> w t -> w t
```

Simple implementation is a world with the highest already issued ID and the list of objects. The operation *createObj* assures that the same ID can not be given again or reused, which is the main concern in temporal databases.

```
data World t = W ID [t]
instance Objects t => Worlds World t where
      createObj t (W i s) = W i’ (t’:s) where
            i’ = nextID i
            t’ = putID i’ t
      getObj i (W j os) = head os' where
            os' = filter ((=i).getID) os
      destroyObj i (W j os) = W j os' where
            os' = filter ((/= i) . getID) os
      perform f i (W i' s) = W i' [if isID i x then f x else x|x<-s]
```

So far, only a snapshot picture of the world is defined. A temporal database shall operate through the history of the objects. This can be represented as a series of world snapshots.

4. EXAMPLE : CADASTRE PARCELS ARE LIQUIDS

Now, we have enough apparatus to analyze the most interesting part of the figure1: lifestyle of fission (splitting) and fusion (merging). The first candidate are liquid objects - fluids.

```
class Fluids f where
      fluidObj  :: Quant -> f
      content   :: f -> Quant
      preds     :: f -> [ID]
      addPreds  :: [ID] -> f -> f
      takeOut   :: Quant -> f -> f
      split     :: Quant -> ID -> f -> f
      merge     :: ID -> ID -> f -> f
```

The last two operations are independent from object datatype and defined as:

```
instance (Worlds World f, Fluids f) => Fluids (World f) where
      split q i w = perform kill i w1 where
            w1 = putObj o1 (putObj o2 w)
            o1 = addPreds ps1 (fluidObj q)
            o2 = addPreds ps1 (takeOut q (getObj i w))
```

```
          ps1 = [i] ++ preds (getObj i w)
     merge i j (W i1 ts) = perform kill i (perform kill j w1) where
          w1 = createObj o1 (W i1 ts)
          o1 = addPreds ps1  (fluidObj (qi + qj))
          ps1 = [i,j] ++ getPreds(getObj i w) ++ getPreds (getObj j w)
          qi = content (getObj i (W i1 ts))
          qj = content (getObj j (W i1 ts))
```

Although it may look strange, the lifestyle of cadastre parcels is similar to those of fluids. The implementations necessary are simple and straightforward:

```
data Parcel = P ID Exist Area Preds
instance Objects Parcel where
     putID i (P j b a p) = P i b a p
     getID (P i b a p) = i
     kill (P i b a p) = P i (kill b) a p
     reincarnate (P i b a p) = P i (reincarnate b) a p
 instance Fluids Parcel where
     fluidObj q = P 0 q True []
     preds (P i cont b ps) = ps
     addPreds is (P i cont b ps) = P i cont b (is++ps)
     content (P id cont b p) = cont
     takeOut q (P id cont b p) = fluidObj (cont - q)
```

So, where are *split* and *merge*? These operations are already fully defined in *Fluids*. It is possible to apply them without any restrictions on any object which is instance of *Fluids* class, including the *Parcel* datatype. It appears that the concept of lifestyles allows a high rate of code-reusability leading to clearer designs of spatio-temporal databases.

5. CONCLUSIONS

It has been shown that functional programming language is excellent formalization tool for complex design of temporal databases based on finite number of simple operations (temporal constructs). The most important temporal features are modeled without introducing the notion of time explicitly. The design process is simplified with the assumption that specific object classes have similar lifestyles. A lifestyle comprises set of common temporal operations a class of objects can undergo. A small, runnable example shows that cadastre parcels correspond with the rather strange world of liquids.

LITERATURE

Al-Taha, K. and R. Barrera (1994). Identities through Time. International Workshop on Requirements for Integrated Geographic Information Systems, New Orleans, Louisiana.

Clifford, J. and A. Croker (1988). Objects in Time. Database Engineering **7**(4): 189-196.

Frank, A.U. et al. (1997). Gofer as used at GeoInfo (TU Vienna). GeoInfo Series Vol. 12, Dept. of Geoinformation, TU Vienna.

Hayes, P. J. (1985). Naive Physics I: Ontology for Liquids. Formal Theories of the Common-sense World. J. R. Hobbs and R. C. Moore. Norwood, NJ, Ablex Publishing: 71-107.

Jones, M. P. (1994). Gofer - Functional Programming Environment, Geoinformation, TU Vienna.