VIII Brazilian Symposium on GeoInformatics, Campos do Jordão, Brazil, November 19-22, 2006, INPE, p. 109-128.

TerraHS: Integration of Functional Programming and Spatial Databases for GIS Application Development

Sérgio Souza Costa[1], Gilberto Câmara[1], Danilo Palomo[1]

[1]Divisão de Processamento de Imagens (DPI) – Instituto Nacional de Pesquisas Espaciais (INPE)
Av dos Astronautas, 1758 – 12227-001 – São José dos Campos – SP – Brasil
{scosta,gilberto, danilo}@dpi.inpe.br

Abstract. *Recently, researchers in GIScience argued about the benefits on using functional programming for geospatial application development and prototyping of novel ideas. This paper presents an application that interfaces a functional language with a spatial database. It enables developing GIS applications development in a functional language, while handling data are in a spatial database. We used this application develop a Map Algebra, that shows the benefits on using this paradigm in GIScience. Our work shows there are many gains in using a functional language, especially Haskell, to write concise and expressive GIS applications. The TerraHS application allows a good compromise between the expressive power of a functional language, and the data handling facilities of an imperative language.*

1 Introduction

Recent, research in GIScience proposes to use functional programming for geospatial application development [Frank and Kuhn 1995; Frank 1997; Frank 1999; Medak 1999; Winter and Nittel 2003]. Their main argument is that many of theoretical problems in GIScience can be expressed as algebraic theories. For these problems, functional languages enable fast development of rigorous and testable solutions [Frank and Kuhn 1995]. However, developing a GIS in a functional language is not feasible, since many parts needed for a GIS are already avaliable in imperative languages such as C++ and Java. This is especially true for spatial databases, where applications such as PostGIS/PostgreSQL offer a basic support for spatial data management. It is unrealistic to develop such support using functional programming.

It is easier to benefit from functional programming for GIS application development if we build an application on top of an existing spatial database programming environment. This work presents TerraHS, an application that enables developing geographical applications in a functional language, using the data handling provided by TerraLib. TerraLib is a C++ library that supports different spatial database management systems, and that includes many spatial algorithms. As a result, we get a combination of the good features of both programming paradigms.

This paper describes the TerraHS application. We briefly review the literature on functional programming and its use for GIS application development in Section 2. We describe how we built TerraHS in Section 3. In Section 4, we show the use of TerraHS for developing a Map Algebra.

2 Brief Review of the Literature

2.1 Functional Programming

Functional programming is a programming paradigm that considers that computing is evaluating off mathematical functions. Functional programming stresses functions, in contrast to imperative programming, which stresses changes in state and sequential commands [Hudak 1989]. Recent functional languages include Scheme, ML, Miranda and Haskell. TerraHS uses the Haskell programming language. The Haskell report describes the language as:

> "*Haskell is a purely functional programming language incorporating many recent innovations in programming language design. Haskell provides higher-order functions, non-strict semantics, static polymorphic typing, user-defined algebraic datatypes, pattern-matching, list comprehensions, a module system, a monadic I/O system, and a rich set of primitive datatypes, including lists, arrays, arbitrary and fixed precision integers, and floating-point numbers*" [Jones 2002].

The next section provides a brief description of the Haskell syntax. This description will help the reader to understand the essential arguments of this paper. For detailed description of Haskell see [Jones 2002], [Peyton Jones, Hughes et al. 1999] and [Thompson 1999].

2.2 A Brief Tour of the Haskell Syntax

Functions are the core of Haskell. A simple example is a function that which adds its two arguments:

```
add :: Integer → Integer → Integer
add x y =  x + y
```

The first line defines the `add` function. It takes two `Integer` values as input and produces a third one. Functions in Haskell can also have generic (or polymorphic) types. For example, the following function calculates the length of a generic list, where `[a]` is a list of elements of a generic type `a`, `[]` is the empty list, and `(x:xs)` is the list composition operation:

```
length :: [a] → Integer
length [] = 0
length (x:xs) = 1 + length xs
```

This definition reads "`length` is a function that calculates an integer value from a list of a generic type `a`. Its definition is recursive. The length of an empty list is zero. The length of a nonempty list is one plus the length of the list without its first element".

The user can define new types in Haskell using a `data` declaration, which defines a new type, or the `type` declaration, which redefines an existing type. For example, take the following definitions:

```
type Coord2D = (Double, Double)
data Point      = Point Coord2D
data Line2D     = Line2D [Coord2D]
```

In these definitions, a `Coord2D` type is a shorthand for a pair of `Double` values. A `Point` is a new type that contains one `Coord2D`. A `Line2D` is a new type that

contains a list of `Coord2D`. One important feature of Haskell lists is that they can be defined by a mathematical expression similar to a set notation. For example, take the expression:

```
[elem | elem <- (domain map) , (predicate elem obj)]
```

It reads "the list contains the elements of a map that satisfy a predicate that compares each element to a reference object". This expression could be used to select all objects that satisfy a topological operator ("*all roads that cross a city*"). Haskell includes *higher-order* functions. These are functions that have other functions as arguments. For example, the `map` higher-order function applies a function to a list, as follows:

```
map   :: (a→b) → [a] → [b]
map f  []      =  []
map f (x:xs)   =  f x : map f xs
```

This definition can reads as "take a function of type `a→b` and apply it recursively to a list of `a`, getting a list of `b`". Haskell supports overloading using *type classes*. A definition of a *type class* uses the keyword `class`. For example, the type class `Eq` provides a generic definition of all types that have an equality operator:

```
class Eq a where
(==)  :: a → a → Bool
```

This declaration reads "a type `a` is an instance of the class `Eq` if it defines is an overloaded equality `(==)` function." We can then specify instances of type class `Eq` using the keyword `instance`. For example:

```
instance Eq Coord2D where
((x1,x2) == (y1,y2))  =  (x1 == x2 && y1 == y2)
```

Haskell also supports a notion of *class extension*. For example, we may wish to define a class `Ord` which *inherits* all the operations in `Eq`, but in addition includes comparison, minimum and maximum functions:

```
class (Eq a) => Ord a  where
  (<), (<=), (>=), (>)  :: a → a → Bool
  max, min              :: a → a → a
```

2.3 Functional Programming and GIS

Many recent papers propose using functional languages for GIS application develepment [Frank and Kuhn 1995; Frank 1997; Frank 1999; Winter and Nittel 2003]. Frank and Kuhn [1995] show the use of functional programming languages as tools for specification and prototyping of Open GIS specifications. Winter and Nittel [2003] apply a formal tool to writing specifications for the Open GIS proposal for coverages. Medak [1999] develops an ontology for life and evolution of spatial objects in an urban cadastre. To these authors, functional programming languages satisfy the key requirements for specification languages, having expressive semantics and allowing rapid prototyping. Translating formal semantics is direct, and the resulting algebraic structure is extendible. However, these works do not deal with issues related to I/O and to database management. Thus, they do not provide solutions applicable to real-life problems. To

apply these ideas in practice, we need to integrate functional and imperative programming.

2.4 Integration of Functional and Imperative Languages

The integration functional and imperative languages is discussed in Chakravarty [2003], who presents the *Haskell 98 Foreign Function Interface (FFI),* which supports calling functions written in C from Haskell and vice versa. However, functions written in imperative languages can contain side effects. To allow functional languages to deal with side effects, Wadler [1990] proposed *monads* for structuring programs written in functional language. The use of monads enables a functional language to simulate an imperative behavior with state control and side effects [Thompson 1999]. Jones [2005] presents many crucial issues about interaction of functional languages with the external world, such as I/O, concurrency, exceptions and interfaces to libraries written in other languages. In this work, the author describes a Haskell web server as a case study. These works show of the integration between these two programming styles. However, none of these works deals with geoinformation systems. On the next section we present an application that integrates programs written in Haskell with spatial databases and allows fast and reliable GIS application development.

3 TerraHS

This section presents TerraHS, a software application which enables developing geographical applications using in functional programming using data stored in a spatial database. TerraHS links the Haskell language to in the TerraLib GIS library. TerraLib is a class library written in C++, whose functions provide spatial database management and spatial algorithms. TerraLib is free software [Vinhas and Ferreira 2005]. TerraHS links to the TerraLib functions through the *Foreign Function Interface* [Chakravarty 2003] and a function set written in C language, which performs the TerraLib functions. The Figure 1 shows its architecture.

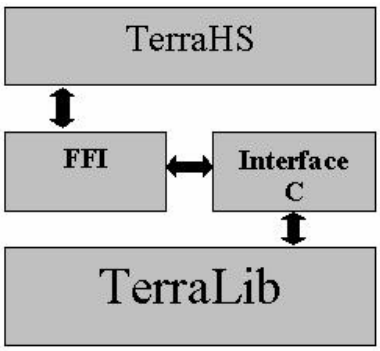

Figure 1 TerraHS Architecture

TerraHS includes three basic resources for geographical applications: *spatial representations, spatial operations* and *database access*. The next sections present them.

3.1 Spatial Representations

3.1.1 Vector data structures

Identifiable entities on the geographical space, or *geo-objects*, such as cities, highways or states are usually represented in vector data structures, as point, line and polygon. These data structures represent an object by one or more pairs of Cartesian coordinates. TerraLib represents coordinate pairs through the *Coord2D* data type. In TerraHS, this type is a tuple of real values.

```
type Coord2D = (Double, Double)
```

The type Coord2D is the basis for all the geometric types in TerraHS, namely:

```
data Point      = Point Coord2D
data Line2D     = Line2D [Coord2D]
type LinearRing = Line2D
data Polygon    = Polygon [LinearRing]
```

The *Point* data type represents a point in TerraHS, and is a single instance of a *Coord2D*. The *Line2D* data type represents a line, composed of one or more segments and it is a vector of *Coord2Ds* [Vinhas and Ferreira 2005]. The *LinearRing* data type represents a closed polygonal line. This type is a single instance of a *Line2D*, where the last coordinate is equal to the first [Vinhas and Ferreira 2005]. The *Polygon* data type represents a polygon in TerraLib, and it is a list of *LinearRing*. Other data types include:

```
data PointSet   = PointSet [Point]
data LineSet    = LineSet [Line2D]
data PolygonSet = PolygonSet [Polygon]
```

3.1.2 Cell-Spaces

TerraLib supports cell spaces. Cell spaces are a generalized raster structure where each cell stores a more than one attribute value or as a set of polygons that do not intercept one another. A cell space enables joint storage of the entire set of information needed to describe a complex spatial phenomenon. This brings benefits to visualization, algorithms and user interface [Vinhas and Ferreira 2005]. A cell contains a bounding box and a position given by a pair of integer numbers.

```
data Cell = Cell Box Integer Integer
data Box  = Box Double Double Double Double
```

The *Box* data type represents a bounding box and the *Cell* data type represents one cell in the cellular space. The *CellSet* data type represents a cell space.

```
data CellSet = CellSet [Cell]
```

3.1.3 Spatial Operations

TerraLib provides a set of spatial operations over geographic data. TerraHS provides function that use those algorithms. We used Haskell type classes [Shields and Jones 2001; Chakravarty 2004] to define the spatial operations using polymorphism. These topologic operations can be applied for any combination of types, such as point, line and polygon.

```
class TopologyOps a b where
```

```
disjoint :: a → b → Bool
intersects :: a → b → Bool
touches  ::  a → b → Bool
…
```

The TopologyOps class defines a set of generic operations, which can be instantiated to several combinations of types:

```
instance TopologyOps Polygon Polygon
instance TopologyOps Point Polygon
instance TopologyOps Point Line2D
…
```

3.2 Database Access

One of the main features of TerraLib is its use of different object-relational database management systems (OR-DBMS) to store and retrieve the geometric and descriptive parts of spatial data [Vinhas and Ferreira 2005]. TerraLib follows a layered model of architecture, where it plays the role of the middleware between the database and the final application. Integrating Haskell with TerraLib enables an application developed in Haskell to share the same data with applications written in C++ that use TerraLib, as shown in Figure 2.

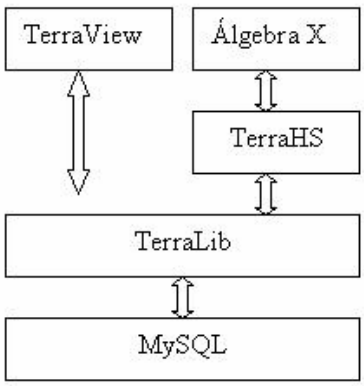

Figure 2 - Using the TerraLib to share a geographical database, adapted from Vinhas e Ferreira (2005).

A TerraLib database access does not depends on a specific DBMS and uses an abstract class called *TeDatabase* [Vinhas and Ferreira 2005]. In TerraHS, the database classes are algebraic data types, where each constructor represents a subclass.

```
data Database = MySQL String String String String
   | PostgreSQL String String String String
```

A TerraLib *layer* aggregates spatial information located over a geographical region and that share the same attributes. A layer is identifier in a TerraLib database by its name [Vinhas and Ferreira 2005].

```
type LayerName = String
```

In TerraLib, a *geo-object* is an individual entity that has geometric and descriptive parts, composed by:

- **Identifier**: identifies a *geo-object*.

```
data ObjectId = ObjectId String
```

- **Attributes**: this is the descriptive part of a geo-object. An attribute has a name (*AttrName*) and a value (*Value*).

```
type AttrName = String
data Value = StValue String| DbValue Double
           |InValue Int | Undefined
data Atribute = Atr (AttrName, Value)
```

- **Geometries**: this is the spatial part, which can have different representations.

```
data Geometry = GPt Point | GLn Line2D | GPg Polygon
              |GCl Cell   | GPtS PointSet (…)
```

A *geo-object* in TerraHS is a triple:

```
data GeObject = GeoObject (ObjectId, [Atribute], Geometry)
```

The *GeoDatabases* type class provides generic functions for storage, retrieval of geo-objects from a spatial database.

```
class GeoDatabases a where
open :: a → IO (Ptr a)
close :: (Ptr a) → IO ()
retrieve :: (Ptr a) → LayerName → IO [GeObject]
store ::(Ptr a) → LayerName → [GeObject] → IO Bool
errorMessage :: (Ptr a) → IO String
```

These operations will then be instantiated to a specific database, such as `mySQL` or `PostgreSQL`. Figure 3 shows an example of a TerraLib database access program.

```
host = "sputnik"
user = "Sergio"
password = "terrahs"
dbname = "Amazonia"
main:: IO()
main = do
   -- accessing TerraLib database
   db <- open (MySQL host user password dbname)
   -- retrieving a geo-object set
   geos <- retrieve db "cells"
   geos2 <- op geos – op is a manipulation operation
   -- storing a geo-object set
   store db "newlayer" geos2
   close db
```

Figure 3 - Acessing a TerraLib database using TerraHS

4 A generalized map algebra

One of the important uses of functional language for GIS is to enable fast and sound development of new applications. As an example, this section presents a map algebra in a functional language. In GIS, *maps* are a continuous variable or to a categorical classification of space (for example, soil maps). Map Algebra is a set of procedures for

handling maps. They allow the user to model different problems and to get new information from the existing data set. The main contribution to map algebra comes from the work of Tomlin [1983]. Tomlin's model uses a single data type (a map), and defines three types of functions. *Local* functions involve matching locations in different map layers, as in "*classify as high risk all areas without vegetation with slope greater than 15%"*. *Focal* functions involve proximal locations in the same layer, as in the expression "*calculate the local mean of the map values*". *Zonal* functions summarize values at locations in a layer contained in zones defined in another layer. An example is "*given a map of city and a digital terrain model, calculate the mean altitude for each city."*

For this experiment, we use the map algebra proposed in Câmara et al. [Câmara 2005]. The authors describe the design of a map algebra that generalizes Tomlin's map algebra by incorporating topological and directional spatial predicates. In the next section, we describe and implement this algebra.

4.1 The map abstract data type

Our map algebra has two main data types: object set and field. An *object set* is a set of objects represented by points, lines or regions associated with nonspatial attribute. *Fields* are functions that map a location in a spatial partition to a nonspatial attribute. The *map* data type combines both the *object set* data type and the *field* data type. A map is a function *m:: E ® A*, where:

- The domain is finite collection, either a set of cells or a set of objects.
- The range is a set of *attribute values.*

For each geographic element $e \in E$, a *map* returns a value $m(e) = a$, where $a \in A$. A geographical element can represent a location, area, line or point. This definition matches the definition of a *coverage* in Open GIS [OGC 2000]. A coverage in a planar-enforced spatial representation that *covers* a geographical area completely and divides it in spatial partitions that may be either regular or irregular. For retrieving data from a coverage, the Open GIS specification propose describes a discrete function (*DiscreteC_Function),* as shown in Figure 4 below.

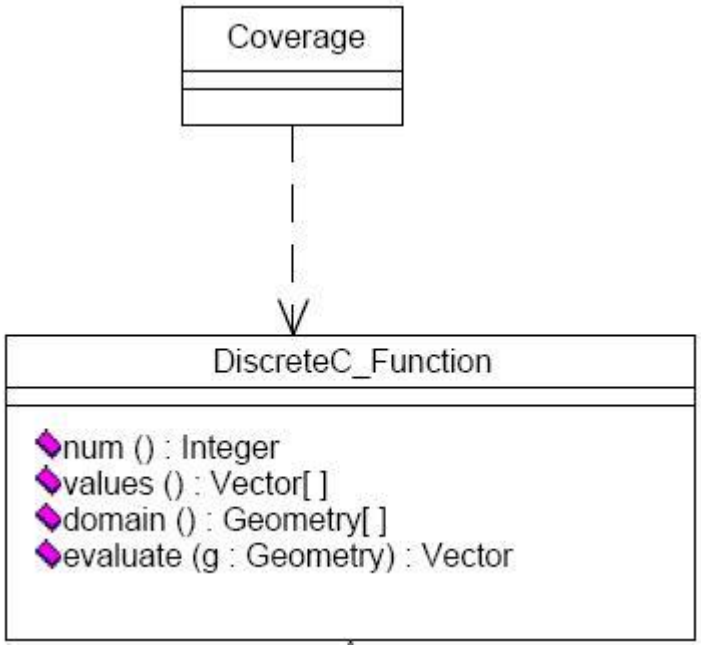

Figure 4 The Open GIS discrete coverage function – source: [OGC 2000].

The *DiscreteCFunction* data type describes a function whose spatial domain and whose range are finite. The domain consists of a finite collection of geometries, where a *DiscreteCFunction* maps each geometry for a value [OGC 2000]. Based on the Open GIS specification, we defined the type class *Maps*. The type class *Maps* generalizes and extends the *DiscreteCFunction* class. Its functions are parameterized on the input type `a` and the output type `b`. It provides the support for the operations proposed by the *DiscreteCFunction*:

```
class Maps map where
   evaluate :: (Eq a, Eq b) => map a b → a → Maybe b
   domain   :: map a b → [a]
   num      :: map a b → Int
   values   :: map a b → [b]
   new_map  :: [a] → (a → b ) → (map a b)
   fun      :: (map a b) → (a → b)
```

The functions is the Maps type class work as follows: (a) `evaluate` is a function that takes a `map` and an input value `a` and produces an output value ("*give me the value of the map at location a*"); (b) `domain` is a function that takes a map and returns the values of its domain; (c) `num` returns the number of elements of the map's domain; (d) `values` returns the values of the map's range. We propose two extra functions: *new_map* and *fun*, as described below.

- `new_map`, a function that returns a new `map` *m,* given a domain and *a coverage function.*
- `fun:` given a `map`, returns its *coverage function*.

We defined the *Map* data type to use the functions of the generic type class *Maps.* The *Map* data type is also parameterized.

```
data Map a b = Map ((a → b), [a])
```

The data type *Map* has two parts:

- A *coverage function* that maps an object of generic type *a* to generic type *b*.
- A domain of objects of the polymorphic type *a*.

The instance of the type class *Maps* to the *Map* data type is shown below:

```
instance Maps Map where
   new_map a f = (Map (f, a))
   evaluate f o
        | (elem o (domain f)) = Just ((fun f) o)
       | otherwise = Nothing
   domain (Map (f, a)) = a
   num f = length (domain f)
   values f  = map (fun f) (domain f)
   fun (Map (f,_)) = f
```

Figure 5 show an example of the *Map* data type.

```
m1 :: (Map String Integer)
m1 = new_map [”ab”,”abc”,”a”] length
values m1
= [2,3,1]
evaluate m1 “ab”
= Just 2
evaluate m1 “ad” -- m1 not contain “ad”
= Nothing
```

Figure 5 Example of use of the *Map* data type.

4.2 Operations

Câmara et al [2005] define two classes of the map algebra operations: nonspatial and spatial. For *nonspatial operations*, the value of a location in the output map is obtained from the values of the same location in one or more input maps. They include logical expressions such as “*classify as high risk all areas without vegetation with slope greater than 15%”, “Select areas higher than 500 meters”, “Find the average of deforestation in the last two years”*, and *“Select areas higher than 500 meters with temperatures lower than 10 degrees”*. *Spatial functions* are those where the value of a location in the output map is computed from the values of the neighborhood of the same location in the input map. They include expressions such as “*calculate the local mean of the map values”* and *“given a map of cities and a digital terrain model, calculate the mean altitude for each city”*. In what follows, we show these operations in TerraHS, using polymorphic data types.

4.2.1 Nonspatial operations

Nonspatial operations are higher-order functions that take one value for each input map and produce one value in the output map, using a first-order function as argument. These include *single argument functions* and *multiple argument functions* [Câmara, Palomo et al. 2005].

```
class (Maps m) => NonSpatialOperations m where
 map_single  :: (b → c) → (m a b) → (m a c)
 map_multiple:: ([b] → c) → [(m a b)] → (m a b)→(m a c)
```

The *map_single* function has two arguments: a *map m* and a first-order function *g*. It returns a new *map,* whose domain contains the same elements of the input map domain. The *coverage function* of the output map is the composition of the *coverage function* of the input map *m* and the first-order function *g*.

```
map_single g m = new_map (domain m)  ( g . (fun m))
```

defines a new map with the same domain		defines the mapping function of the new map

Figure 6 shows an example of a single argument function.

```
values m1
= [2, 4, 12]
m2 = map_single square m1
values m2
= [4, 16, 144]
```

Figure 6 Example of use of the single argument function

The *map_multiple* function has three arguments: a *map* list, a multivalued function and a reference *map*. Given a reference *map,* it applies a multivalued function in *map* list.

```
map_multiple fn mlist mref =
   new_map (domain mref) (\x →  fn (map_r mlist x))
```

defines a new map with the same domain		defines the mapping function of the new map using an auxiliary function

The *map_multiple* function returns a new *map* with a same domain of the reference map and a new coverage function. This function uses the auxiliary function *map_r*. For each element *x* of the reference map, *map_r* applies the multiargument function in the input list of maps to get the output value. It also handles cases where there are multiargument function fails to returns an output value.

```
map_r :: (Maps m) => [(m a b )] → a → [b]
map_r [] _ = []
map_r (m:ms) e = map_r' (evaluate m x)
   where
   map_r' (Just v)  = v : (map_r ms e)
   map_r' (Nothing) = (map_r ms e)
```

Figure 7 shows an example of *map_multiple*. In this example, the *m3 map* is the result of the sum of the *maps m1* and *m2*.

```
values m1
```

```
= [2, 4, 8]
values m2
= [4, 5, 10]
m3 = map_multiple sum [m1, m2] m1
values m3
= [6, 9, 18]
```

Figure 7 - Example of use of *map_multiple*

4.2.2 Spatial Operations

Spatial operations are higher-order functions that use a spatial predicate. These functions combine a selection function and a multivalued function, with two input maps (the reference map and the value map) and an output map [Câmara, Palomo et al. 2005]. Spatial functions generalize Tomlin's focal and zonal operations and have two parts: *selection* and *composition*. For each location in the output map, the *selection function* finds the matching region on the reference map. Then it applies the spatial predicate between the *reference* map and the *value* map and creates a set of values. The *composition function* uses the selected values to produce the result (Figure 8). Take the expression *"given a map of cities and a digital terrain model, calculate the mean altitude for each city"*. In this expression, the *value map* is the digital terrain model and the *reference map* is the map of cities. The evaluation has two parts. First, it selects the terrain values inside each city. Then, it calculates the average of these values.

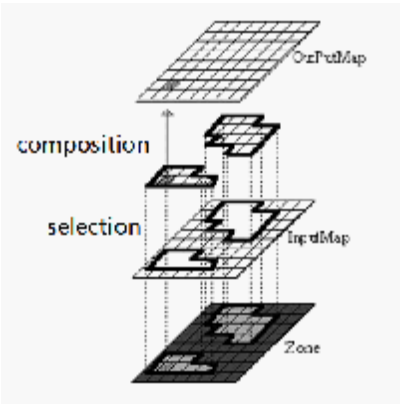

Figure 8. Spatial operations (selection + composition). Adapted from Tomlin [1990].

The implicit assumption is that the geographical area of the *output map* is the same as *reference map*. The type signature of the spatial functions in TerraHS is:.

```
class (Maps m) => SpatialOperations m where
     map_select  :: (m a b) → (a → c→ Bool) → c → (m a b)
     map_compose :: ([b] → b) → (m a b) →  b
     map_spatial :: ([b] → b) → (m a b) → (a → c → Bool)
          → (m c b) → (m c b)
```

The *spatial selection* function selects all elements that satisfy a predicate on a reference object ("*select all deforested areas inside the state of Amazonas*"). It has three arguments: an input *map*, a predicate and a reference element.

```
map_select m pred obj = new_map sel_dom (fun m)
```

```
where
sel_dom = [elem | elem ← (domain m) , (pred elem obj)]
```

This function takes a reference element and an input map. It creates a map that contains all elements of the input map that satisfy the predicate over the reference element. Figure 9 shows an example, where the map consists of a set of points. Then, we select those points that intersect a given line.

```
line= Line2d [Point(1,2),Point(2,2),Point (1,3),Point (0,4)]
domain m1
= [Point(4,5),Point (1,2),Point (2,3),Point (1,3)]
m2 = map_select m1 intersects line
domain m2
= [Point (1,2), Point (1,3)]
```

Figure 9 Example of *map_select.*

The *composition function* combines selected values using a multivalued function. In Figure 10, the `map_compose` function is applied to *map m1* and to the multivalued function *sum*.

```
map_compose f m = (f (values m))
```

```
values m1
= [ 2, 6, 8]
map_compose sum m1
= 16
```

Figure 10 Example of *map_compose.*

The `map_spatial` function combines *spatial selection* and *spatial composition*:

```
map_spatial fn m pred mref = new_map (domain mref)
   (\x → map_compose (map_select m pred x) fn)
```

Map_spatial creates a map whose domain contains the elements of the reference map. To get its coverage function, we apply *map_compose* to the result of the *map_selection*. Figure 12 shows an example.

```
domain m1
= [Point(4,5),Point (1,2),Point (2,3),Point (1,3)]
values m1
= [2,4,5,10]
domain m2
= [(Line2d[Point(1,2),Point (2,2),Point (1,3),Point (0,4)])]
m3 = map_spatial sum m1 intersects m2
values m3 -- 4 + 10
= [14]
```

Figure 12 Example of *map_spatial*

The spatial operation selects all points of *m1* that intersect *m2* (which is a single line). Then, it sums its values. In this case, points (1,2) and (1,3) intersect the line. The sum of their values is 14.

4.3 Application Examples

In the previous section we described how to express the map algebra proposed in Câmara et al. [2005] in TerraHS. In this section we show the application of this algebra to actual geographical data.

4.3.1 Storage and Retrieval

Since a *Map* is generic data type, it can be applied to different concrete types. In this section we apply it to the *Geometry* and *Value* data types available in the TerraHS, which represent, respectively, a region and a descriptive value. TerraHS enables storage and retrieval of a *geo-object* set. To perform a map algebra, we need to convert from a *geo-object* set to a map and vice versa.

```
toMap :: [GeObject] → AttrName → (Map Geometry Value)
toGeObject::(Map Geometry Value)→ AttrName → [GeObject]
```

Given a geo-object set and the name of one its attributes, the `toMap` function returns a `map`. Remember that a *Map* type has one value for each region. Thus, a layer with three attributes it produce three *Maps*. The `toGeObject` function inverts the `toMap` function. Details of these two functions are outside the scope of this paper. Given these functions, we can store and retrieve a map, given a spatial database.

```
retrieveMap::
      Database → LayerAttr → IO (Map Geometry Value)
retrieveMap db (layername, attrname) = do
      db <- open db
      geoset <- retrieve db layername
      let map = toMap geoset attrname
      close db
      return map
```

The `LayerAttr` type is a tuple that represents the layer name and attribute name. The `retrieveMap` function connects to the database, loads a geo-object set, converts these geo-objects into a map, and return this map as its output.

```
storeMap::
   Database→ LayerAttr → (Map Geometry Value)→ IO Bool
storeMap db (layername, attrname) m = do
   let geos = toGeObject map attrname
   db <- open db
   close db
   let status = store db layername geos
   return status
```

The `storeMap` function coverts a map to a geo-object set that will be saved in the database. We can now write a program that reads and writes a *map* in a TerraLib database.

```
host = “sputnik”
user = “Sergio”
password = “terrahs”
dbname = “Amazonia”
main:: IO ()
main = do
   db <- open (MySQL host user pass dbname)
   def_map <- retrieveMap db (“amazonia”,“deforest")
   -- apply a nonspatial operation
   let defclass = map_single classify def_map
   storeMap db (“amazonia”, “defclass”) defclass
```

Figure 13 Retrieving and storing a *Map* from TerraLib Database

4.3.2 Examples of Map Algebra in TerraHS

Since 1989, the Brazilian National Institute for Space Research has been monitoring the deforestation of the Brazilian Amazon, using remote sensing images. We use some of this data as a basis for our examples. We selected a data set from the central area of Pará, composed by a group of highways and two protection areas. This area is divided in cells of 25 x 25 km2, where each cell describes the percentage of deforestation and deforested area (Figure 14).

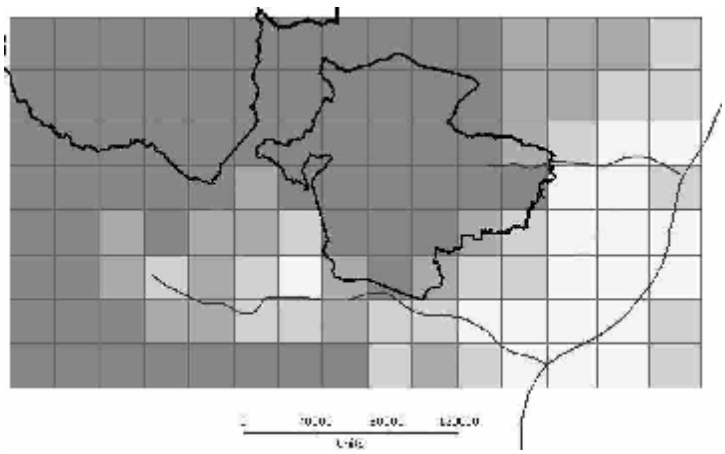

Figure 14 – Deforestation, Protection Areas and Roads Maps (Pará State)

Our first example considers the expression: “*Given a map of deforestation and classification function, return the classified map*”. The classification function defines four classes: (1) dense forest; (2) mixed forest with agriculture; (3) agriculture with forest fragments; (4) agricultural area. This function is:

```
classify :: Value → Value
classify (DbValue v)
   | v < 0.2 = (StValue "1")
   | ((v > 0.2) && (v < 0.5)) = (StValue "2")
   | (v > 0.5) && (v < 0.8) = (StValue "3")
   | v > 0.8 = (StValue "4")
```

We obtain the classified map using the `map_single` operation together with the `classify` function:

```
def_class = map_single classify def_map
```

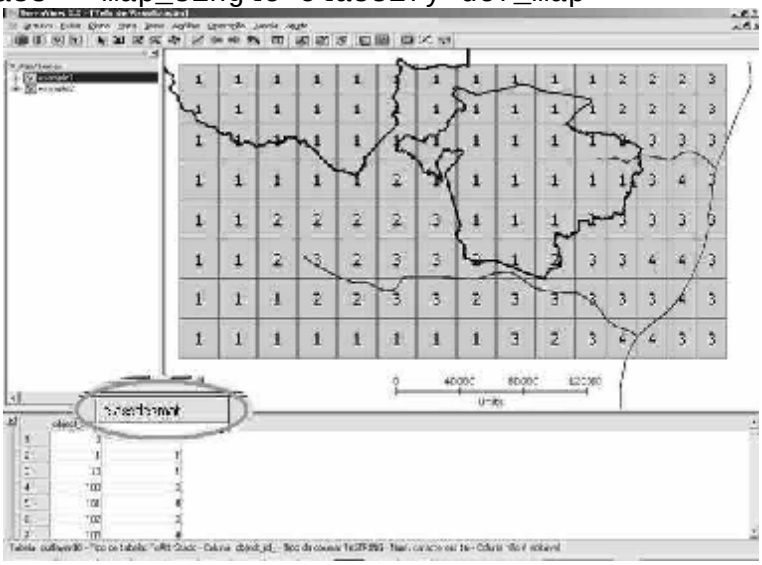

Figure 15 – The classified map

As a second example, we take the expression: "*Calculate the mean deforestation for each protection area*". The inputs are: the deforestation map (`def_map`), a spatial predicate (`within`), a multivalued function (`mean`) and the map of protected areas (`prot_areas`). The output is a deforestation map of the protected areas (`def_prot`) with the same objects as the reference map (`prot_areas`). We use the `map_spatial` higher-order operation to produce the output:

```
def_prot = map_spatial mean def_map within prot_areas
```

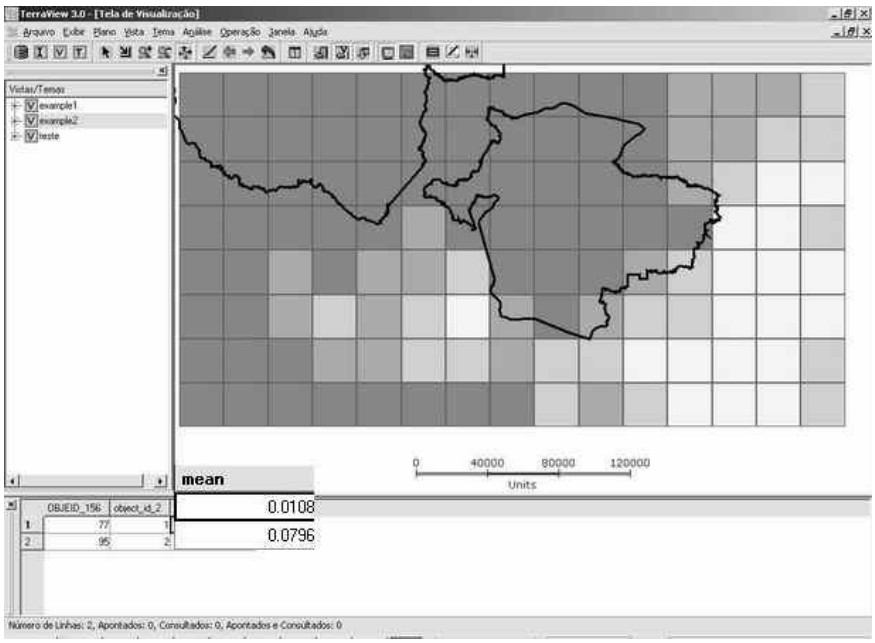

Figure 16 – Deforest mean by protection area

In our third example, we consider the expression: "*Given a map containing roads and a deforestation map, calculate the mean of the deforestation along the roads*". We have as inputs: the deforestation map (`def_map`), a spatial predicate (`intersect`), a

multivalued function (`mean`) and a road map (`roads`). The product is a map with one value for each road. This value is the mean of the cells that intercept this road.

```
road_def = map_spatial mean def_map intersect road_map
```

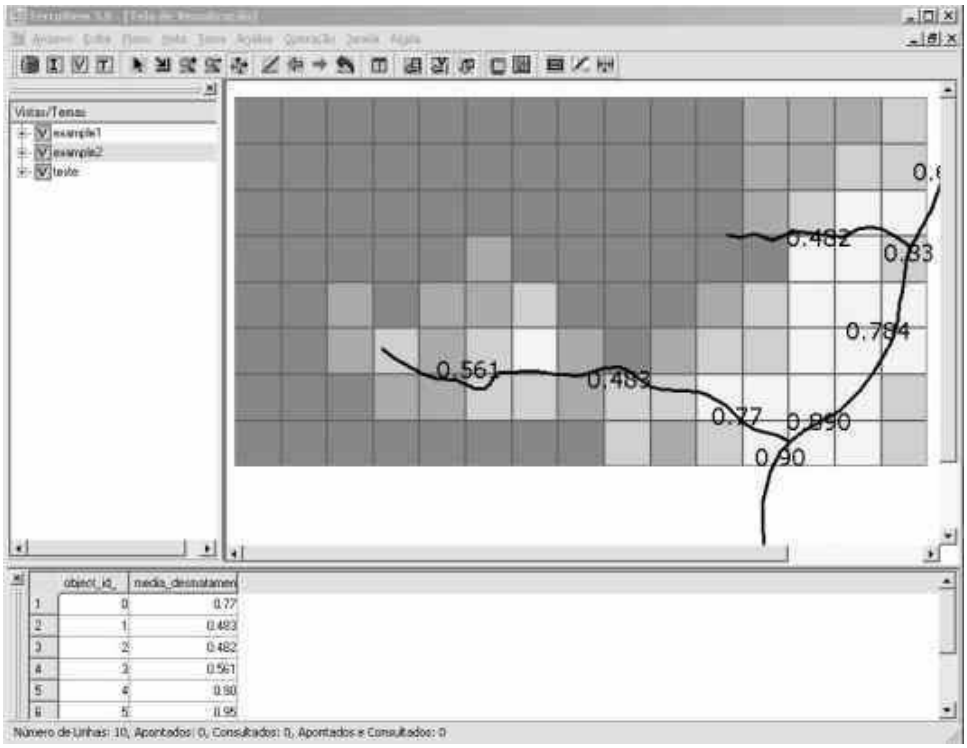

Figure 17 – Deforestation mean along the roads

5 Conclusions

This paper presents the TerraHS application for integrating functional programming and spatial databases. We use TerraHS to develop and validate a map algebra in a functional language. The resulting map algebra is compact, generic and extensible. The example shows the benefits on using functional programming, since it enables a fast prototyping and testing cycle. Table 1 presents the total number of Haskell lines used to develop the map algebra.

Table 1 – Map Algebra in Haskell

	Number of source lines		
	operations	**axioms**	**total**
Data types	6	9	15
Map Algebra	6	10	16
Auxiliary	1	5	6
Total	13	24	37

For comparison purposes, the SPRING GIS [Câmara, Souza et al. 1996] includes a map algebra in the C++ language that uses about 8,000 lines of code. The SPRING

map algebra provides a strict implementation of Tomlin's algebra. Our map algebra allows a more generic set of functions than Tomlin's at less than 1% of the code lines. This large difference comes from the use of the parameterized types, overloading and higher order functions, which are features of the Haskell language. Our work points out that integrating functional languages with spatial database is an efficient alternative in for developing and prototyping novel ideas in GIScience.

References

Câmara, G. (2005). Representação computacional de dados geográficos. Bancos de Dados Geográficos. M. Casanova, G. Câmara, C. Davis, L. Vinhas and G. Ribeiro. Curitiba, MundoGeo Editora**:** 11-52.

Câmara, G., D. Palomo, R. C. M. d. Souza, et al. (2005). Towards a generalized map algebra: principles and data types. VII Workshop Brasileiro de Geoinformática, Campos do Jordão, SBC.

Câmara, G., R. Souza, U. Freitas, et al. (1996). "SPRING: Integrating Remote Sensing and GIS with Object-Oriented Data Modelling." Computers and Graphics **15**(6): 13-22.

Casanova, M., G. Camara, C. Davis, et al., Eds. (2005). Bancos de Dados Geograficos (Spatial Databases). Curitiba, Editora MundoGEO.

Chakravarty, A. P. a. M. (2004). Interfacing Haskell with Object-Oriented Languages. 15th International Workshop on the Implementation of Functional Languages, Lübeck, Germany, Springer-Verlag.

Chakravarty, M. (2003). "The Haskell 98 foreign function interface 1.0: An addendum to the Haskell 98 report."

Frank, A. (1997). Higher order functions necessary for spatial theory development. Auto-Carto 13, Seattle, WA, ACSM/ASPRS.

Frank, A. (1999). One Step up the Abstraction Ladder: Combining Algebras - From Functional Pieces to a Whole. COSIT - Conference on Spatial Information Theory, Springer-Verlag.

Frank, A. and W. Kuhn (1995). Specifying Open GIS with Functional Languages. Advances in Spatial Databases—4th International Symposium, SSD '95, Portland, ME. M. Egenhofer and J. Herring. Berlin, Springer-Verlag. **951:** 184-195.

Hudak, P. (1989). "Conception, evolution, and application of functional programming languages." ACM Comput. Surv. **21**(3): 359-411.

Jones, S. P. (2002). "Haskell 98 Language and Libraries The Revised Report."

Jones, S. P. (2005). "Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell."

Medak, D. (1999). Lifestyles - a new Paradigm in Spatio-Temporal Databases. Department for Geoinformation. Vienna, Technical University of Vienna.

OGC. (2000). "Open GIS Consortium. Topic 6: the coverage type and its subtypes." Retrieved 10/05/2006, 2006, from http://portal.opengeospatial.org/files/?artifact_id=7198.

Peyton Jones, S., J. Hughes and L. Augustsson. (1999). "Haskell 98: A Non-strict, Purely Functional Language." from http://www.haskell.org/onlinereport/.

Shields, M. and S. L. P. Jones (2001). "Object-Oriented Style Overloading for Haskell." Electronic Notes in Theoretical Computer Science **59**(1).

Thompson, S. (1999). Haskell:The Craft of Functional Programming. Harlow, England, Pearson Education.

Tomlin, C. D. (1983). A Map Algebra. Harvard Computer Graphics Conference. Cambridge, MA.

Vinhas, L. and K. R. Ferreira (2005). Descrição da TerraLib. Bancos de Dados Geográficos. M. Casanova, G. Câmara, C. Davis, L. Vinhas and G. Ribeiro. Curitiba, MundoGeo Editora**:** 397-439.

Wadler, P. (1990). Comprehending monads. Proceedings of the 1990 ACM conference on LISP and functional programming %@ 0-89791-368-X. Nice, France, ACM Press**:** 61-78.

Winter, S. and S. Nittel (2003). "Formal information modelling for standardisation in the spatial domain." International Journal of Geographical Information Science **17**: 721--741.