



MINISTÉRIO DA CIÊNCIA E TECNOLOGIA

INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

INTEGRATION OF FUNCTIONAL PROGRAMMING AND SPATIAL
DATABASES FOR GIS APPLICATION DEVELOPMENT

Sérgio Souza Costa

Master Thesis in Applied Computing Science, advised by Dr.
Gilberto Câmara

INPE
São José dos Campos
2006

MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

INPE-

INTEGRATION OF FUNCTIONAL PROGRAMMING AND SPATIAL
DATABASES FOR GIS APPLICATION DEVELOPMENT

Sérgio Souza Costa

Master Thesis in Applied Computing Science, advised by Dr.
Gilberto Câmara

*“O que é que a ciência tem?
Tem lápis de calcular
Que é mais que a ciência tem?
Borracha pra depois apagar”.*

RAUL SEIXAS

*Dedico a minha vó:
MARILIS BASILIA DA CONCEIÇÃO, in memoriam.*

AGRADECIMENTOS

Agradeço a todas pessoas que me ajudaram a vencer mais esta etapa da vida, mais em especial a três pessoas, que foram mais do que fundamentais no alcance desse sonho: ao Gilberto Câmara, que sempre confiou em mim e me ajudou onde eu mais precisei. Ao Miguel, que foi a primeira pessoa, nesse instituto, a acreditar em meu trabalho. E ao meu amigo Paulo Lima, pelo seu apoio nos primeiros passos rumo a esse trabalho.

Agradeço ainda, ao Instituto Nacional de Pesquisas Espaciais – INPE, pela oportunidade de estudos e utilização de suas instalações.

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq, pela concessão de bolsa de Desenvolvimento Tecnológico Industrial (DTI).

A GISPLAN Tecnologia da Informação, pelo auxílio financeiro de um ano de bolsa estágio.

A todos os amigos da Divisão de Processamento de Imagem – DPI, pelo verdadeiro espírito de equipe, onde cito algumas pessoas como: Lúbia, Karine, Gilberto Ribeiro, Ricardo Cartaxo, Leila, João Ricardo, Hilceia, Ana Paula, Isabel e Julio entre outros, que de uma forma ou de outra ajudaram na conclusão desse trabalho.

Aos meus pais, Antonio e Iracema, que me educaram, me apoiaram em todos momentos da vida e ainda por todo amor dado, um eterno obrigado. Aos meus irmãos, Gilberto Aparecido, Maria, Marli, Toninho, Fernando, Rinaldo, Vanda e Marcio, por todo apoio e amizade. E aos meus queridos sobrinhos e cunhados. E a minha estimada vó, in memoriam, Marilis, pela grande mulher que foi e a quem eu devo a bela família que tenho.

Aos meus grandes amigos: Adair, Danilo, Evaldinolia, Karla, Olga, Eduilson, Érica, Joelma, Vantier, Thanisse, Fábio, Missae, Fred e Javier, por sua verdadeira amizade.

Um muito obrigado, a todas as pessoas citadas acima e minhas desculpas a quem eu deixei de citar.

INTEGRAÇÃO DE PROGRAMAÇÃO FUNCIONAL E BANCO DE DADOS ESPACIAIS NO DESENVOLVIMENTO DE APLICATIVOS GEOGRÁFICOS

RESUMO

A pesquisa recente em *geoinformação* indica que há benefícios no uso de programação funcional aplicada ao desenvolvimento de aplicativos geográficos. No entanto, o desenvolvimento completo de um sistema de geoinformação em linguagem funcional não é factível. O acesso a banco de dados geográfico exige um grande conjunto de operações de entrada e saída, de difícil implementação em linguagens funcionais. Essa dissertação apresenta um aplicativo que integra uma linguagem funcional (Haskell) com banco de dados espacial (TerraLib). Esta integração permite o desenvolvimento, em uma linguagem funcional, de aplicativos geográficos que manipulem dados em um banco de dados espacial. Esse aplicativo foi usado no desenvolvimento de uma Álgebra de Mapas, que mostra os benefícios do uso desse paradigma em *geoinformação*. Nosso trabalho mostrou que existem muitas vantagens no uso de uma linguagem funcional, especialmente Haskell, no desenvolvimento de aplicativos geográficos mais expressivos e concisos. Combinando **Haskell** e **TerraLib**, nós permitimos o uso de programação funcional em problemas reais, e tornamos o Haskell uma ferramenta ainda mais amplamente usada no desenvolvimento de aplicativos geográficos.

INTEGRATION OF FUNCTIONAL PROGRAMMING AND SPATIAL DATABASES FOR GIS APPLICATION DEVELOPMENT

ABSTRACT

Recently, researchers in GIScience argued about the benefits on using functional programming for geospatial application development and prototyping of novel ideas. However, developing an entire GIS in a functional language is not feasible. Support for spatial databases requires a large set of I/O operations, which are cumbersome to implement in functional languages. This thesis 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 in a spatial database. We used this application to develop a Map Algebra, which 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. Combining **Haskell** and **TerraLib** enables the use of functional programming to real-life GIS problems, and is a contribution to make Haskell a more widely used tool for GIS application development.

TABLE OF CONTENTS

	Page
1. INTRODUCTION.....	17
2. THEORETICAL FOUNDATIONS	20
2.1 Functional Programming.....	20
2.2 A Brief Tour of the Haskell Syntax	22
2.3 I/O in Haskell using Monads	26
2.4 Foreign Language Integration.....	30
2.5 Algebraic Specification and Functional Programming	33
2.6 Functional Programming for Spatial Databases and GIS Applications.....	35
3. TERRAHS.....	38
3.1 Introduction	38
3.2 System Architecture	39
3.3 Spatial Data Types.....	43
3.4 Spatial Operations.....	47
3.5 Spatial Database Access.....	50
4. A GENERALIZED MAP ALGEBRA IN TERRAHS.....	53
4.1 Introduction	53
4.2 Tomlin's Map Algebra: a brief review.....	53
4.3 Research challenges for map algebra.....	55
4.4 Spatial predicates as a basis for Map Algebra.....	56
4.5 The Open GIS Coverage in Haskell.....	56
4.6 Map Algebra Operations	59
4.7 Map Algebra Operations in TerraHS	60
4.8 Application Examples	64
4.8.1 Storage and Retrieval	64
4.8.2 Examples of Map Algebra in TerraHS.....	65
4.9 Discussion of the Results.....	69
5. CONCLUSION AND FUTURE WORKS.....	71
REFERENCES	73

FIGURE LIST

Figure 3.1 - TerraHS: General View.....	38
Figure 3.2 - TerraHS Architecture.....	39
Figure 3.3 – Using the disjoint operation.....	41
Figure 3.4 – Haskell Hello World Program	42
Figure 3.5 – A simple TerraHS program.	42
Figure 3.6 – Second TerraHS program.	43
Figure 3.7 - Vector representation - source: Casanova {, 2005 #20}.....	43
Figure 3.8 - Example of the use of the vector data types.....	44
Figure 3.9 - A cell space graphic representation	45
Figure 3.10 – A cell space in TerraHS.....	45
Figure 3.11 – Example of use geometry data type	46
Figure 3.12 - Example of use GeoObject data type.....	46
Figure 3.13 - Topologic operations.....	48
Figure 3.14 - Centroid operation	48
Figure 3.15 - Example of overlay operation	49
Figure 3.16 – Example of metric operations	49
Figure 3.17 - Using the TerraLib to share a geographical database, adapted from Vinhas e Ferreira (2005).....	50
Figure 3.18 - TerraLib database drivers - source: Vinhas and Ferreira {, 2005 #18}	50
Figure - 3.19 - Acessing a TerraLib database using TerraHS	51
Figure 4.1-Spatial operations (selection + composition).Adapted from {Tomlin, 1990 #129}	60
Figure 4.2 – Deforestation, Protection Areas and Roads Maps (Pará State)	66
Figure 4.3 – The classified coverage	67
Figure 4.4 – Deforestation mean by protection area.....	68
Figure 4.5 – Deforestation mean along the roads	69

CHAPTER 1

INTRODUCTION

Developing geographic information systems is a complex enterprise. GIS applications involve data handling, algorithms, spatial data modeling, spatial ontologies and user interfaces. Each of these presents unique challenges for GIS application development. Broadly speaking, there are three main parts on a GIS application. Spatial databases provide for storage and retrieval of spatial data. User interfaces include the well-established WIMP methaphor (windows, icons, mouse and pointers), as well as novel techniques such as direct manipulation and virtual reality. Between the interface and the database, we find many spatial algorithms and spatial data manipulation. These include techniques such as map algebra, spatial statistics, location-based services and dynamical modeling.

The diversity of data manipulation techniques, as well as the various ways of combining them, is an intimidating problem for GIS developer. Therefore, to build successful GIS application we must resort to the well-known “divide and conquer” principle. It is best to break a complex system into modular parts and design each part separately. If these parts are built in a proper way, they can be combined in different ways to build efficient and successful GIS applications. As Meyer(1999) defines, a component is *“a software element that must be usable by developers who are not personally known to the component’s author to build a project that was not foreseen by the component’s author.”*

Research in Geographic Information Science has shown than many spatial data manipulation problems can be expressed as algebraic theories (Tomlin, 1990; Egenhofer e Herring, 1991; Frank e Kuhn, 1995; Güting, T. De Ridder *et al.*, 1995; Frank, 1997; Erwig, Güting *et al.*, 1999; Frank, 1999; Medak, 1999; Güting, Bohlen *et al.*, 2003; Winter e Nittel, 2003). These algebraic theories formalize spatial components in a rigorous and generic way. This brings a second problem: how to translate an algebraic

specification into a programming language. Ideally, the resulting code should as expressive and generic as the original algebraic specification. In practice, limits of the chosen programming language interfere in the translation. For example, the Java programming language is unable to express generic data types. Similar problems arise in other languages, such as C++ and C# (Hughes, 1989).

As an answer to the challenges of translation of algebraic specifications into computer languages, there has been a growing interest in functional languages. Functional programming is so called because a program consists entirely of functions (Hughes, 1989). The main program itself is written as a function which receives the program's input as its argument and delivers the program's output as its result. Features of modern functional languages also include list-processing functions, higher-order functions, lazy evaluation and support for generic programming. These features allow a close association between abstract specifications and computer code. The resulting programs are more concise and more accurate. Since the modules are smaller and more generic, they can be reused more widely (Hughes, 1989).

Recently, researchers in GIScience have argued about the benefits of functional programming for geospatial application development and prototyping of novel ideas (Frank e Kuhn, 1995; Frank, 1997; Frank, 1999; Medak, 1999; Winter e Nittel, 2003). Among the proposed benefits of functional programming for GIS is the ability to build complex systems from small parts (Frank e Kuhn, 1995). Each of these small parts is expressed as an algebra and developed in a rigorous and testable fashion. The resulting algebras are abstract building blocks which can be combined to create more complex solutions.

However, developing an entire GIS in a functional language is not feasible. Support for spatial databases needs a large set of I/O operations, which are cumbersome to implement in functional languages. Event-driven user interfaces are better implemented with callback protocols and are difficult to specify formally. It is also not practical to double services already available in imperative languages such as C++ and

Java. This is especially true for spatial databases, where applications such as PostGIS/PostgreSQL offer a good support for spatial data management.

Our hypothesis is that *to integrate functional programming and spatial databases for GIS application development, we should build a functional GIS on top of an existing spatial database support*. We then use each programming paradigm in the most efficient fashion. We rely on imperative languages such as C++ to provide spatial database support and we use functional programming for building components that provide data manipulation algorithms.

To assess our hypothesis, we have built **TerraHS**, an application development system that enables developing geographical applications in the Haskell functional language. **TerraHS** uses the data handling abilities 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 styles. Our hypothesis is tested by developing a Map Algebra using TerraHS.

This thesis describes our work to corroborate our hypothesis. We briefly review the literature on functional programming and its use for GIS application development in Chapter 2. We describe how we built TerraHS in Chapter 3. In Chapter 4, we show the use of TerraHS for developing a Map Algebra. We close the work (in Chapter 5) by pointing out future lines of research.

CHAPTER 2

THEORETICAL FOUNDATIONS

This chapter presents the foundations for our work. Given that functional programming may be unfamiliar to the reader, we present a brief tour of the Haskell syntax, to help in understanding of our work in Chapters 3 and 4. We also present important references that link functional programming and GIS.

2.1 Functional Programming

Almost all programs currently developed use imperative programming. Imperative programming uses assignment of values to variables and explicit control loops, and is supported by languages such as C++, Java, Pascal, and FORTRAN. In this work, we highlight functional programming, which considers that computing is evaluating mathematical functions. Functional programming stresses functions, in contrast with imperative programming, which stresses changes in state and sequential commands (Hudak, 1989). Backus (1978) presents a comparison between the functional and imperative programming styles. According to Backus, imperative languages are versions of the Von Neumann computer:

“.. use variables to imitate the computer's storage cells; control statements elaborate its jump and test instructions; and assignment statements imitate its fetching, storing, and arithmetic”.

Consider an imperative program that calculates the sum of all members in a list, in written in the imperative language C:

```
sum = 0;
for (int i = 0; i < n ; i++)
    sum = sum + list[i];
```

This program has several properties:

- Its statements act on an invisible "state" according to complex rules.

- It is not hierarchical. Except for the right side of the assignment statement, it does not compose complex entities from simpler ones. (Larger programs, however, often do.)
- It is repetitive. One must mentally execute it to understand it.
- It computes word-at-a-time by repetition (of the assignment) and by change (of variable *i*).
- Part of the data, *n*, is in the program; thus it lacks generality and works only for lists of length *n*.

The functional equivalent (in Haskell) does not have any variable updates.

```
sum [] = 0
sum (x:xs) = x + (sum xs)
```

This program version uses two important features in functional language: recursion and pattern matching. Pattern matching is the act of checking for the presence of a given pattern. Standard patterns include variables, constants, the wildcard pattern, patterns for tuples, lists, and algebraic constructors. The first line says the sum of an empty list is 0. In the second line, $(x:xs)$ stands for a list as a *tuple*. The head of the list is *x* and the rest of the list is *xs*. The second line reads: “*The sum of a non-empty list is the sum of the first member with the rest of the list*”.

Functional programming is different from imperative programming. Functional programming contains no side effects and no assignment statements. A function produces a side effect if it changes some state other than its return value. For example, a function that prints something to the screen has side effects, since it changes the value of a global variable. Backus (1978) considers that a functional program has important advantages over its imperative counterpart. A function program: (a) acts only on its arguments; (b) is hierarchical and built from simpler functions; (c) is static and nonrepetitive; (d) handles whole conceptual units; (e) employs idioms that are useful in many other programs.

LISP (McCarthy, 1963) was the first functional programming language. Recent functional languages include Scheme, ML, Miranda and Haskell. In this work we will use the Haskell as programming language in the TerraHS software application presented in Chapter 4. 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, nonstrict 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” (Peyton Jones, 2002).

Haskell is a *typeful* programming language. It offers a rich type system, and enforces strong type checking. It is also a safe language. According to Pierce, “a *safe* language protects its own abstractions and makes it impossible to shoot yourself in the foot while programming” (Pierce, 2002). For detailed description of Haskell, see (Peyton Jones, 2002), (Peyton Jones, Hughes *et al.*, 1999) and (Thompson, 1999).

2.2 A Brief Tour of the Haskell Syntax

This section provides a brief description of the Haskell syntax. This description will help the reader to understand the essential arguments of this work. For the rest of this section, we use Hudak *et al.*(1999), (Peyton Jones, 2001) and (Daume, 2004).

2.2.1 Functions and Lists

Functions are the core of Haskell. Consider the `add` function, shown below. It takes two `Integer` values as input and produces a third one. The first line defines its signature and the second defines its implementation.

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

Lists are a commonly used data structure in Haskell. The list `[1,2,3]` in Haskell is shorthand for the list `1:2:3:[ ]`, where `[ ]` is the empty list and `:` is the infix operator

that adds its first argument to the front of its second argument (a list). Functions in Haskell can also have generic (or polymorphic) types, and most list functions are polymorphic. The following function calculates the length of a generic list, where `[a]` is a list of members 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 member*”. The definition also shows the pattern matching features of Haskell. The `length` function has two expressions, which are evaluated in the order they are declared.

Haskell lists can also be defined by a mathematical expression similar to a set notation:

```
[ x | x <- [0..100], prime x ]
```

This expression defines “*the list of all prime numbers between 0 and 100*”. This is similar to the mathematical notation

```
{ x | x ∈ [0..100] ∧ prime (x) }
```

This expression is useful to express spatial queries. Take the expression:

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

It reads “*the list contains the members of a map that satisfy a predicate that compares each member to a reference object*”. This expression could be used to select all objects that satisfy a topological operator (“*all roads that cross a city*”). A further example is the following implementation of *quicksort*:

```

quicksort :: [a] → [a]
quicksort [] = []
quicksort (x:xs) =  quicksort [y | y <- xs, y<x ]
                    ++ [x]
                    ++ quicksort [y | y <- xs, y>=x]

```

2.2.2 Data Types

Haskell has strict type checking. Each value has an associated type. Haskell provides built-in atomic types: `Integer`, `Char`, `Bool`, `Float` and `Double`. From these types one can define types such as `Integer→Integer` (functions mapping `Integer` to `Integer`), `[Integer]` (homogeneous lists of integers) and `(Char,Integer)` (character, integer pairs).

The user can define new types in Haskell using the 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 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`. Type definitions can be recursive. Here is a simple declaration of an algebraic data type and a function accepting an argument of the type, which shows the basic features of algebraic data types in Haskell:

```

data Tree a = Leaf a | Branch (Tree a) (Tree a)
size :: Tree a → Integer
size (Leaf x) = 1
size (Branch r l) = 1 + size r + size l

```

2.2.3 Higher-Order Functions

An important feature of Haskell is *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 read as “*take a function of type $a \rightarrow b$ and apply it recursively to a list of a , getting a list of b* ”. One example is applying a function that doubles the members of a list:

```
map (double) [1, 2, 3, 4] ⇒ [2, 4, 6, 8]
```

The `map` higher-order function is useful for GIS operations, since many of the GIS operations are transformations on lists. A simple example is a function that translates all the coordinates of a line.

```
type Line = [(Double, Double)]
translate :: (Double, Double) → [Line] → [Line]
translate (x,y) lin = map add (x,y) lin
                  where add (x1,y1) (x2,y2) = ((x1+x2),(y1+y2))
```

Note the auxiliary `add` function defined by the `where` keyword.

2.2.4 Overloading and Type Classes

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 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*. An example is 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
```

Type classes are the most unusual feature of Haskell's type system. Type classes have proved extraordinarily convenient in practice (Hudak, Peyton Jones *et al.*, 2007), since they are extensible and type-safe.

2.3 I/O in Haskell using Monads

The final purpose of a spatial database application is to cause a side effect, such writing a new record in a database. However, side effects cause problems in functional languages (Peyton Jones, 2001). Things like “*print a string to the screen*” or “*read data from a file*” are not functions in the pure mathematical sense. Therefore, Haskell gives them another name: *actions* (Daume, 2004). Actions are part of a more general notion of *computation*. A computation is more general notion than a function. It includes issues such as dealing with failures and telling about success. A computation also includes the issues of sequencing. In essence, we need to represent success and failure. Also, we need a way to combine two successes. Finally, we need to be able to augment a previous success (if there was one) with some new value (Daume, 2004). We can fit this all into a class as follows:

```
class Computation c where
  success  :: a -> c a
  failure  :: String -> c a
  sequence :: c a -> c b -> c b
  combine  :: c a -> (a -> c b) -> c b
```

`Computation` is a type class which is parameterized on a type `c`. It has four actions. The `success` function takes a value of type `a` and returns it wrapped up by `c`,

representing a successful computation. The `failure` function takes a `String` (error message) and returns a computation representing a failure. The `sequence` function is used to support a sequence of unrelated actions. It enables a computation of type `c a` to be followed by a computation of type `c b` and still keep the correct result type. For example, we can build a sequence of actions that print a character on a screen and then save the same character on a file. The `combine` function enables using the result of an action to be used as the input for another action. We may read a character on a screen and then print the resulting value. It has two inputs: a previous computation (namely, `c a`) and a function which maps the value of that computation (the `a`) to a new computation (`c b`). It returns a new computation (`c b`) (Daume, 2004).

The notion of computation is expressed more formally in Haskell by the idea of *monads*. Wadler (1990) proposed *monads* for structuring programs written in functional languages, based in the work of Eugenio Moggi (1991). The use of monads enables a functional language to simulate an imperative behavior with state control and side effects (Thompson, 1999). To define monads in Haskell, we use a shortened version of Computation:

```
class Monad m where
    return :: a -> m a
    fail   :: String -> m a
    (>>)   :: m a -> m b -> m b
    (>>=)  :: m a -> (a -> m b) -> m b
```

The functions `return` and `fail` match `success` and `failure` of the Computation class. The symbols `(>>)` and `(>>=)` match our definitions of `sequence` and `combine`, discussed above. An important example of a monad is the I/O monad, which deals with computations that affect the “state” of the world. The key notion is the idea of a type `IO a`. A computation of type `IO a` does some I/O, then produces a value of type `a`. For example, the function `getChar` has type `IO Char`, since it does some I/O and returns a type `Char`. The function `main` (the main program loop in Haskell) has type `IO ()`, since it does some I/O and returns nothing. Thus we can express the type `IO a` as:

```
type IO a = World -> (a, World)
```

This type definition says that a value of type `IO a` is a function that, when applied to an argument of type *World*, delivers a new *World* with a result of type `a` (Peyton Jones, 2001). For instance, a function that prints something to the screen causes an effect in this *World* and returns it. In this case the *World* is a screen, but it could be a file, a database or some other I/O device.

```
helloworld :: IO ()
helloworld = print "Hello World"
```

Imagine that we want to write a little more sophisticated “hello world” program in Haskell. Our function takes a user’s input and then prints the given input. This needs the I/O monad, which is an instance of a Monad for the I/O type:

```
instance Monad IO where
    return a = ...
    m >=> k  = ...
    fail s   = IOError (userError s)
```

Monadic I/O treats a sequence of I/O commands as a computation that interacts with the outside world. The program specifies an action, which the compiler turns into real I/O. For example, we can use the `combine` function (`>=>`) to get a line from the screen and print it:

```
printName :: IO ()
printName = getLine >=> print
```

The `combine` function (`>=>`) is also described as “bind”. When the compound action (`a >=> f`) is performed, it performs action `a`, takes the result, applies `f` to it to get a new action, and then performs that new action (Peyton Jones, 2001). Suppose that now we want to precede the `getLine` function by a message to user using by the `print` function. We can’t use the combinator (`>=>`), because (`>=>`) expects a function with two arguments as its second argument, and not a function with just one argument. We must use the `sequence` function (`>>`). This function simply consumes the argument of the first action, throws it away, and carries out the next action. Now we can write.

```
printName :: IO ()
printName = print "Enter a string" >> getLine >=> print
```

Haskell has syntactic support for monadic programming, called the *do notation*. This notation is a simple translation to the ($\gg=$, $\gg$) functions. The syntax is much more convenient, so in practice everyone uses do notation for I/O-intensive programs in Haskell. But it is just notation! (Peyton Jones, 2001). Using the do notation we can write `printName` as follows

```
printName :: IO ()
printName = do
    print "Enter a string"
    line <- getLine
    print line
```

A further example of a monad is the `Maybe` monad, which deals with computations that might not succeed.

```
data Maybe a = Nothing
              | Just a

instance Monad Maybe where
    return x = Just x
    (Just x) >>= f = f x
    Nothing  >>= _ = Nothing
    fail _   = Nothing
```

The `Maybe` monad is a polymorphic algebraic data type. In case of failure, it uses the `Nothing` constructor; in case of success, it uses the `Just` constructor, with a value of type `a`. Suppose we want to write a function that finds a value in a given list and that treats errors gracefully. This function can be written as:

```
find :: (Eq a) => a -> [a] -> Maybe a
find _ [] = Nothing
find x (y:ys)
    | (x == y) = Just y
    | otherwise = find x ys
```

In resume, the concept of monads is extremely powerful, and allows a rigorous definition of the notion of computation. Using monads, functional programming can be extended to include models of computation formerly typical of imperative programming (Jones e Wadler, 1993).

2.4 Foreign Language Integration

One feature that many applications need is the ability to call procedures written in some other language from Haskell, and preferably vice versa (Hudak, Peyton Jones *et al.*, 2007). Interaction with other languages is crucial to any programming language. For example, GIS applications make extensive use of spatial data management that is offered by applications such as PostGIS/PostgreSQL written in C language. It is unrealistic to develop such support using functional programming; instead, we want to make it easy to call them. Chakravarty (2003) proposed the *Haskell 98 Foreign Function Interface (FFI)*, which supports calling functions written in C from Haskell and the order reversed. The FFI treats C as a lowest common denominator: once you can call C you can call almost anything else (Hudak, Peyton Jones *et al.*, 2007). Suppose a function that plots a point in the screen written a C language.

```
void plotPoint (double x, double y);
```

We can call this procedure from Haskell, under the FFI proposal:

```
foreign import ccall plotPoint :: Double → Double → IO()
```

As usual, we use the *IO monad* in the result type of `plotPoint` to tell that `plotPoint` may perform I/O, or have some other side effect. However, some foreign procedures may have purely functional semantics. For example, consider the *disjoint* topologic operation applied to two points.

```
bool disjoint (double x1,double y1, double x2, double y2);
```

This function has no side effects. In this case it is tiresome to force it to be in the *IO monad*. So the Haskell FFI allows one to use the *unsafe* keyword, and omit the “IO” from the return type, thus (Peyton Jones, 2001):

```
foreign import ccall unsafe disjoint :: Double →  
    Double → Double → Double → Bool
```

Haskell types can be used in arguments and results just in types such as *Int*, *Float*, *Double*, and so on. However, most real program use memory references or pointers as an abstract object representation, structured types or arrays. For instance, consider the following structure data type.

```

struct Point {
    double x y;
}

```

The *Point* structure contains two attributes, *x* and *y*, which represent the Cartesian coordinates. The *disjoint* topologic operation takes two arguments of the *Point* data type structure.

```

bool disjoint_p (Point* p1, Point* p2);

```

Haskell provides the *Ptr* data type to represent a pointer to an object, or an array of objects. For instance, consider the following definition:

```

type PointPtr = Ptr ()

```

This is equivalent in C language to:

```

typedef void *PointPtr;

```

However, this approach rules out distinguishing between different objects. To improve this, we can define *typed pointers* (Peyton Jones, 2001; Chakravarty, 2005).

```

data Point = Point
type PointPtr = Ptr Point

```

As a result, pointers to objects are typed; we can then use different type class instances for different objects (Chakravarty, 2004). We can call the *disjoint* topologic operation (*disjoint_p*) under the FFI:

```

foreign import ccall unsafe disjoint_p :: PointPtr →
    PointPtr → Bool

```

To simplify code development, there are some tools available, called foreign function interface preprocessors for Haskell. These preprocessors simplify the task of interfacing Haskell programs with external libraries written in C Language. Some examples are (Yakeley, 2006):

- **GreenCard:** Green Card is a foreign function interface preprocessor for Haskell, simplifying the task of interfacing Haskell programs to external libraries (which are normally exposed via C interfaces).

- **HaskellDirect:** HaskellDirect is an Interface Definition Language (IDL) compiler for Haskell, which helps interfacing Haskell code to libraries or components written in other languages (C). An IDL specification specifies the type signatures and types expected by a set of external functions. One important use of this language neutral specification of interfaces is to specify COM (Microsoft's Component Object Model) interfaces, and HaskellDirect offers special support for both using COM objects from Haskell and creating Haskell COM objects.
- **C→Haskell:** A lightweight tool for implementing access to C libraries from Haskell.
- **HSFFIG:** Haskell FFI Binding Modules Generator (HSFFIG) is a tool that takes a C library include file (.h) and creates Haskell Foreign Functions Interface import declarations for items (functions, structures, etc.) defined in the header.
- **Kdirect:** A tool to simplify linking C libraries to Haskell. It is less powerful than HaskellDirect, but easier to use and more portable.

These preprocessors deal with the interface between Haskell and C. However, there are many libraries implemented in object-oriented languages, such as C++ or Java. Some authors teach how to map object-oriented languages to Haskell (Peyton Jones, Meijer *et al.*, 1998; Finne, Leijen *et al.*, 1999; Meijer e Finne, 2000; Shields e Jones, 2001; Chakravarty, 2004). These works point out that some features of object-oriented languages cannot be encoded directly in Haskell, such as inheritance and overloading. They present a way of calling of methods from object-oriented languages in a syntactically convenient type-safe manner. The authors encode class inheritance via *phantom types* and *type class*. Object-oriented overloading is encoded via *name-mangling*, *closed class* and multiparameter type class. Chakravarty (2004) presents how to bind Haskell and Objective-C. Meijer (2000) discusses integration between Haskell and Java.

2.5 Algebraic Specification and Functional Programming

In this section we discuss the relation between functional programming and algebraic specification. Guttag (1978) presents algebraic specification as tool to define abstract data types. The specification consists of a set of functions, where the values of the type are created and inspected only by calls to these functions. This allows the implementation to be changed without any changes to the external type interface. Algebraic specifications consist of three parts: a type, a set of operations and axioms, that describe how the operations are apply in specific data type.

Functional programming languages are convenient to translate algebraic specifications into testable code (Frank e Kuhn, 1995; Frank e Medak, 1997). Functional languages express the semantics of abstract data types directly, an essential property for formal specification languages (Frank e Kuhn, 1995). To explain this point, consider a data type in Haskell:

```
data Point = Pt Double Double
```

`Point` is a concrete data type, where `Pt` defines a constructor to this data type. A constructor is an operator that builds new objects. In this case the constructor takes two arguments of type `Double`. We can instantiate this definition inside a Haskell program:

```
pt1 = Pt 20.3 50.5
```

Suppose we want to change the `Pt` constructor to take a single argument:

```
data Point = Pt (Double, Double)
```

This change will impact other programs that reference the `Point` data type, since this definition is implementation-dependent. Abstract data types hide the implementation from the user, and its data is only accessible through a set of operations. In Haskell, we can define these abstract data type as a *type class*:

```
class Points a where
  createPoint :: Double → Double → a
  getX :: a → Double
  getY :: a -> Double
```

In this example, the type class `Points` has three operations: one constructor (`createPoint`) and two observers (`getX` and `getY`). An observer is an operator that lets you examine an object without changing it. We can then instantiate the type class `Points` and create a concrete data type:

```
data Point = Pt Double Double
instance Points Point where
    createPoint x y = Pt x y
...
```

The same operations could be defined to other implementation of the `Point` data type:

```
data Point = Pt (Double, Double)
instance Points Point where
    createPoint x y = Pt (x, y)
...
```

An algebraic specification can be applied to more than one abstract type, creating multisorted algebras. Frank (1999) and Lin (1998) propose using multisorted algebras for specifying GIS applications. We can implement multisorted algebras in Haskell, using a *type class* with multiple parameters. Suppose that we want to describe an interface to deal with points without specifying how their coordinates are expressed (such as integers or floats). We can define a type class `Points` with multiple parameters:

```
class Points p a where
    createPoint :: a → a → p
    getX :: p → a
    getY :: p → a
```

This type class can be instantiated to different data types:

```
data DbPoint = DbPt Double Double
instance Points DbPoint Double
...
```

or

```
data InPoint = InPt Int Int
instance Points InPoint Int
...
```

2.6 Functional Programming for Spatial Databases and GIS Applications

Research in Geographic Information Science has shown that many spatial data manipulation problems can be expressed as algebraic theories (Tomlin, 1990; Egenhofer e Herring, 1991; Frank e Kuhn, 1995; Güting, T. De Ridder *et al.*, 1995; Erwig, Güting *et al.*, 1999; Güting, Bohlen *et al.*, 2003). This leads to a growing interest in using functional languages for GIS application development. (Frank, 1997; Frank, 1999; Medak, 1999; Winter e Nittel, 2003; Frank, 2005)

Frank and Kuhn (1995) show the use of functional programming languages as tools for specification and prototyping of Open GIS specifications. The authors discuss the pros and cons of using functional languages for writing specification. The main advantage is their expressiveness: formal specifications can be translated directly into executable programs. and extensibility. The authors consider that functional languages lack certain desirable properties: they are not designed for a formal verification of specifications, nor for version management, or for documentation and cooperation in teams. However, there are no reasons why such tools could not be constructed for a functional language. As example, the authors presented the *Point* data type specification in a functional language. The specifications focus on equality operations, leaving additional operations on points (such as a distance) unspecified.

Winter and Nittel (2003) present the results and experiences of applying a functional language as formal tool to writing specifications for the Open GIS proposal for *coverages*. The authors compare the functional language with UML in specification of geographic systems. The work shows how to map the Open GIS Coverage UML classes to Haskell classes and function declarations. This mapping allows the authors to discuss consistency, correctness and completeness of the Open GIS coverage specification. Medak (1999) develops an ontology for life and evolution of spatial objects in an urban cadastre. Based in category theory, (Frank, 2005), demonstrates how to extend and generalize Tomlin's Map Algebra to apply uniformly for spatial, temporal, and spatio-temporal data. In his view, a map layer and a time series are functors, which map operations on single values (*sum*, *mean* and so on) to operations on

layers and time series. 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.

As an example, consider the algebra of moving objects proposed by Güting et al (2003). They define a basic type *moving point* (*mpoint*) as a mapping between a temporal reference and a spatial location:

$$mpoint = time \rightarrow point$$

A moving point can be used to describe how a car moves along a road. This basic type can have several functions, including:

at: $mpoint \times time \rightarrow point$

minvalue: $mpoint \rightarrow point$

start, stop: $mpoint \rightarrow time$

duration: $mpoint \rightarrow real$

trajectory: $mpoint \rightarrow line$

A simple implementation of the *mpoint* data type in Haskell is:

```
type Mpoint = [(Time, Point)]
```

where *Time* and *Point* are suitably defined classes. Supposing the pairs (*Time*, *Point*) are ordered by *Time*, then coding these functions in Haskell is simple. For example, take the *trajectory* function:

```
trajectory :: MPoint → Line
trajectory mp = [snd (pr) | pr ← mp ]
```

The *snd* function extracts the second member of the pair (*Time*, *Point*). Thus, it is straightforward to move from an algebraic specification to its equivalent code in Haskell. The simplicity of this translation has led to many authors to conclude that Haskell is suitable for geospatial application development (Frank e Kuhn, 1995; Frank, 1997; Frank, 1999; Medak, 1999; Winter e Nittel, 2003). 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, as we describe in the next chapter.

CHAPTER 3

TERRAHS

3.1 Introduction

In this Chapter, we present the design and implementation of TerraHS, a software application that enables developing geographical applications in functional programming using data stored in a spatial database. TerraHS links the Haskell language to 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 e Ferreira, 2005). TerraHS can be compiled in Linux or Windows platforms, where the requirements in both platforms are: *ghc*-6.4.1 (Glasgow Haskell compiler) or later, TerraLib-3.1.0, MySQL-4.1 and *gcc*-3.4.2. In Windows and in the Linux, the libraries TerraLib and MySQL should be compiled for gcc GNU compiler. That restriction is necessary because *ghc* linker requires the libraries provided by the *gcc* compiler, and does not support other compilers, such as Microsoft's Visual C++. TerraHS includes access to three basic resources for geographical applications: *spatial representations*, *spatial operations* and *spatial databases*, as shown in Figure 3.1.

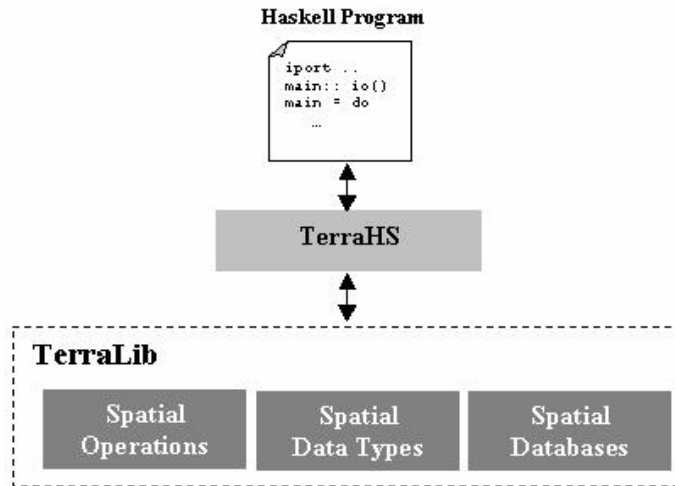


Figure 3.1 - TerraHS: General View

We present the TerraHS architecture in Section 3.2. In sections 3.3, 3.4 and 3.5 we describe the main TerraHS resources.

3.2 System Architecture

TerraHS links to TerraLib using the *Foreign Function Interface* (FFI) (Chakravarty, 2003) and to additional code written in C (*TerraLibC*), which maps the FFI to TerraLib methods. In the Figure 3.2, lighter colors represent the parts built in this work and darker colors represent the existing components.

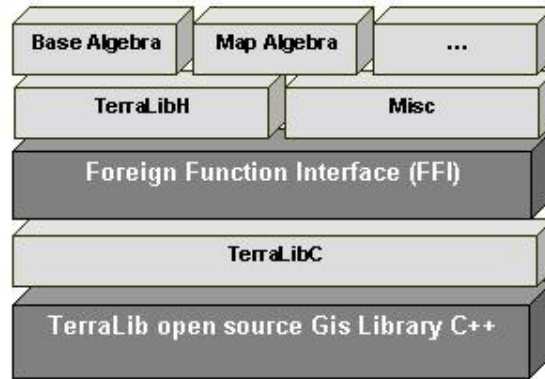


Figure 3.2 - TerraHS Architecture

Lower layers provide basic services over which upper layer services are implemented. In the bottom layer, TerraLib supports different spatial database management and many spatial algorithms. In the second layer, *TerraLibC* maps the TerraLib C++ methods to the Haskell FFI. In the third layer, the FFI enables calling the *TerraLibC* functions from Haskell. The two last layers contain a set of Haskell *modules*, which develop functional applications in Haskell using TerraLib. Haskell programmers normally use modules to build large programs. We group the modules in two main directories: TerraHS and Algebras. The TerraHS directory contains the following subdirectories:

- **TerraLibH**: contains the modules that map TerraLib C++ classes to Haskell data types and functions, `TeGeometry.hs`, `TeDatabase.hs` and so on.

- **Misc:** contains the modules that provide auxiliary functions to TerraHS, such as `Time.hs` and `Generic.hs`.

The `Algebras` directory contains algebras to support other Haskell programs. There is a main algebra, called `base algebra`, which provides basic spatial database management and spatial operations. In this work we have also built a map algebra, presented in Chapter 4. Other algebras can be implemented and shared in this directory, increasing the scope of TerraHS.

3.2.1 Mapping TerraLib Classes to Haskell

In section 2.4, we showed how to structure pointers in Haskell. This is especially important in libraries for GIS application that use complex data structures. In this section, we show how to map TerraLib classes and Haskell data types. For instance, consider the following TerraLib class.

```
class TePoint {
    public:
        TePoint (double x, double y ) {...}
        double getX () ;
        double getY () ;
        ...
}
```

The simplest mapping from Haskell to C uses *phantom types*:

```
data TePoint = TePoint      --- A phantom type
type TePointPtr = Ptr TePoint -- pointer to TePoint class
make_tePoint :: Double → Double → TePointPtr
getX :: TePointPtr → Double
getY :: TePointPtr → Double
```

The *TePoint* type is called a *phantom type* because it does not appear as a value on the right side. This approach was used in other works (Jones, 1995; Finne, Leijen *et al.*, 1999; Meijer e Finne, 2000; Chakravarty, 2004). In this work, we prefer *nonphantom types*, which are more adequate in Haskell programs:

```
data TePoint = TePoint (Double,Double) -- non-phantom type
type TePointPtr = Ptr Point -- pointer to TePoint class
```

Nonphantom types, in a similar way to C++ classes, use constructors to build a new object. Based on this, we propose a *type class* for mapping pointers to algebraic data types and vice versa.

```
class Pointer a where
  -- | map haskell type to a pointer
  toPointer :: a → (Ptr a)
  -- | map a pointer to a Haskell type
  fromPointer :: (Ptr a) → a
```

Using the previous example, we have the following instance:

```
instance Pointer TePoint where
  toPointer TePoint (x, y) = make_tepoint x y
  fromPointer ptr = TePoint((getX ptr ),(getY ptr ) )
```

The *Pointer type class* is instanced to other TerraLib data types, like:

```
instance Pointer TeLine2D where ..
instance Pointer TePolygon where ..
...
```

The *pointer type class* is used internally in TerraHS. Consider the following topologic function mapped from TerraLib:

```
tedisjoint :: TePointPtr → TePointPtr → Bool
```

This function uses a pointer from *TePoint* class. However, in Haskell, it is more interesting to use full Haskell types than pointers. Thus, we set up the following new operations that use Haskell types:

```
disjoint :: TePoint → TePoint → Bool
disjoint p1 p2 = tedisjoint (toPointer p1) (toPointer p2)
```

The *disjoint* operation can be used directly in Haskell programs:

```
pt1 = TePoint (23.4, 45.6 )
pt2 = TePoint (5.6, 78.3 )
d = disjoint pt1 pt2
⇒ True
```

Figure 3.3 – Using the disjoint operation.

The above example is just illustrative. Topologic operations will be presented in the section 3.4.1.

3.2.2 Compiling TerraHS Programs

A Haskell program has a main function. A simple program is shown in Figure 3.4:

```
main:: IO()
main = do
  print "Hello World !!"
```

Figure 3.4 – Haskell Hello World Program

The first line has the main function. This is the entry point to the Haskell program, similar to `main()` in C programs. In Haskell, `main` takes nothing and returns an *IO* monad. To compile a Haskell program with `ghc`, you use a command such as:

```
ghc -o program program.hs
```

Before we start a TerraHS program, it is necessary to import the modules that include the TerraLib data types, provided by the `TerraHS.TerraLib` module, Figure 3.5.

```
import TerraHS.TerraLib
main:: IO()
main = do
  pt1 = (TePoint (2,3))
  print pt1
  => TePoint (2,3)
```

Figure 3.5 – A simple TerraHS program.

After TerraHS is installed, a TerraHS program can be compiled using the following command:

```
ghc -fghlasgow-exts -ffi -o tehspr tehspr.hs -package TerraHS-0.1
```

In Haskell, the libraries are divided into *packages*. For example, the *base package* contains the *Prelude* module. This package is available any extra flags; it will be automatically loaded the first time they are needed. Other packages can be loaded using the `-package` flag, as the TerraHS *package*. To use a spatial operation, we need to import the `Algebras.Base.Operations` module, Figure 3.6. This program is compiled the same way.

```

import TerraHS.TerraLib
import Algebras.Base.Operations
main:: IO()
main = do
    let pt1 = TePoint (2,3)
    let pt2 = TePoint (7,3)
    print distance pt1 pt2
    => 5

```

Figure 3.6 – Second TerraHS program.

This program defines two points, and then it prints the distance between them. The data types and operations from TerraHS are covered in sections: 3.3, 3.4 and 3.5.

3.3 Spatial Data Types

TerraHS provides support to the basic types in Terralib. In its current version, it supports vector data structures and cell-space. This data types are accessible to Haskell program by importing the `TerraHS.TerraLib` module.

3.3.1 Vector Data Structures

Identifiable entities on the geographical space, or geo-objects, such as cities, highways or states are usually represented by vector data structures, such as *point*, *line* and *polygon*. These data structures represent an object by one or more pairs of Cartesian coordinates, as shown in Figure 3.7.

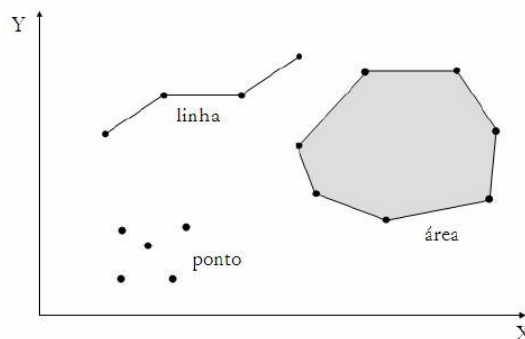


Figure 3.7 - Vector representation - source: Casanova (2005).

TerraLib represents coordinate pairs through the *TeCoord2D* data type. In TerraHS, this type is a tuple of real values.

```
type TeCoord2D = (Double, Double)
```

The *TeCoord2D* type is the basis for all the geometric types in TerraHS, namely:

```
data TePoint      = TePoint TeCoord2D
data TeLine2D     = TeLine2D [TeCoord2D]
type TeLinearRing = TeLine2D
data TePolygon    = TePolygon [TeLinearRing]
```

The *TePoint* data type represents a point in TerraHS, and is a single instance of a *TeCoord2D*. The *TeLine2D* data type represents a line, composed of one or more segments and it is a vector of *TeCoord2Ds* (Vinhas e Ferreira, 2005). The *TeLinearRing* data type represents a closed polygonal line. This type is a single instance of a *TeLine2D*, where the last coordinate is equal to the first (Vinhas e Ferreira, 2005). The *TePolygon* data type represents a polygon in TerraLib, and it is a list of *TeLinearRing*. Other data types include:

```
data TePointSet   = TePointSet [TePoint]
data TeLineSet    = TeLineSet [TeLine2D]
data TePolygonSet = TePolygonSet [TePolygon]
```

Figure 3.8 shows examples of vector data types.

```
pt = TePoint (4.2, 5.7)
ln = TeLine2D [(2,5),(3,4), (5,6) ]
pol = TePolygon [ (TeLine2D [(TePoint (4.2, 5.7) ), ...] ) ]
...
```

Figure 3.8 - Example of the use of the vector data types

3.3.2 Cell-Spaces

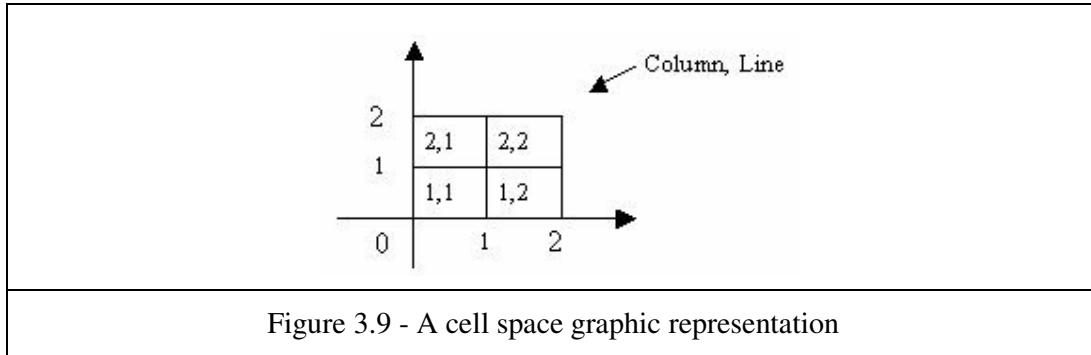
TerraLib supports cell spaces. Cell spaces are a generalized raster structure where each cell stores a more than one attribute value or as a set of polygons that do not intercept one another. A cell space enables joint storage of the entire set of information needed to describe a complex spatial phenomenon.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

```
data TeCell = TeCell TeBox Integer Integer
data TeBox  = TeBox Double Double Double Double
```

The TeBox data type represents a bounding box and the TeCell data type represents one cell in the cellular space. The TeCellSet data type represents a cell space.

```
data TeCellSet = TeCellSet [TeCell]
```

Consider the following cell space:



This cell space in TerraHS is implemented as:

```
cels = TeCellSet [ (TeCell (TeBox 0 0 1 1) 1 1),
  (TeCell (TeBox 0 1 1 2) 2,1),
  (TeCell (TeBox 1 1 2 2) 1,2),
  (TeCell (TeBox 1 0 1 2) 2,2)]
```

Figure 3.10 – A cell space in TerraHS

Each cell has a unique identification and a unique reference to its position inside the cell space. It also has a set of attributes. Since these attributes are the same as those used by the geo-object data type, they will be discussed in the next section.

3.3.3 Geo-Object Data Type

In TerraLib, a geo-object is an individual entity that has geometric and descriptive parts.

Identifier

Identifiers are used to give to each geo-object a unique identity to distinguish a geo-object in TerraLib database. In the TerraLib an identifier is represented by string.

```
data ObjectId = ObjectId String
```

Attributes

Attributes are the descriptive part of a geo-object. An attribute has a name (*AttrName*) and a value (*Value*). We support different data types for values.

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

The same geo-object can contain different data types for values. For instance, a city can contain some attributes as: (`"Name", (StValue "São José dos Campos")`) , (`"Population", (InValue 580000)`) and (`"IDH", DbValue 0.81`)).

Geometry

Geometry is the spatial part, which can have different representations. The possible representations were defined in the section 3.3.1.

```
data TeGeometry = GPt TePoint | GLn TeLine2D
                | GPg TePolygon | GC1 TeCell (...)
```

Figure 3.11 show an example of the use Geometry data type.

```
geol = GPt ( TePoint (4.2, 5.7) )
geo2 = GLn ( TeLine2D [(2,5),(3,4), (5,6) ] )
...
```

Figure 3.11 – Example of use geometry data type

Definition

A geo-object in TerraHS is a triple:

```
data GeObject = GeObject (ObjectId, [Attribute], [Geometry])
```

Figure 3.12 shows an example of the GeObject data type.

```
attr1 = Attr ("Attr1", (InValue 1) )
attr2 = Attr ("Attr2", (InValue 2) )

geol = GPg (Polygon [ ( Line2d[(4,5),(3,2),... ] ) ]
go = GeObject (ObjectId "1", [attr1,attr2], [geol] )
```

Figure 3.12 - Example of use GeObject data type

3.4 Spatial Operations

TerraLib provides a set of spatial operations over geographic data. Vinhas (2005) groups them in five classes:

- **Topological relationships among vector geometries:** relationships include *touch*, *contain*, *within*, *covered by*.
- **Metric operations:** area calculation, length or perimeter and geometries distance.
- **Building new geometries:** *buffer*, *centroid* and convex hull.
- **Combining geometries:** include *difference*, *union*, *intersection* or *symmetrical difference*.
- **Map algebra:** 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 core of TerraHS includes the above, except map algebra. We used Haskell type classes (Shields e Jones, 2001; Chakravarty, 2004) to define the spatial operations using polymorphism. They are accessible in Haskell by importing the `Algebras.Base.Operations` module. In the next sections we present the core TerraHS spatial functions.

3.4.1 Topologic Operations

Topologic operations can be applied for any combination of types, such as point, line and polygon. They are grouped in the `TopologyOps` type class:

```
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 TePolygon TePolygon
instance TopologyOps TePoint TePolygon
instance TopologyOps TePoint TeLine2D

```

Figure 3.13 shows an example of topologic operations.

```

poll = TePolygon[(TeLine2d [(1,1),(1,3),(3,3),(3,1),(1,1)])]
pol2 = TePolygon[(TeLine2d [(2,2),(2,4),(4,4),(4,2),(2,2)])]
test = intersect poll pol2
⇒ True

```

Figure 3.13 - Topologic operations

3.4.2 Centroid Operation

Centroid is the term given to the center of an area, region, or polygon. It is described as an x,y coordinate.

```

class Centroid a where
  centroid :: a -> TeCoord2D

```

In the same way, centroid operation can be instantiated to several geometric types:

```

instance Centroid TePolygon where
instance Centroid TeLine2D where
instance Centroid TePointSet where
...

```

Figure 3.14 shows an example of a centroid operation.

```

poll = TePolygon [(TeLine2d [(1,1),(1,3),(3,3),(3,1),(1,1)])]
center = centroid poll
⇒ TePoint (2,2)

```

Figure 3.14 - Centroid operation

3.4.3 Overlay Operations

Overlay operations, or set operations, is other important class of operations provided in TerraHS. These operations were grouped in the *Overlay type class*:

```

class Overlay a where
  union          :: [a] → [a] → [a]
  intersection    :: [a] → [a] → [a]
  difference      :: [a] → [a] → [a]

```

We provide in TerraHS-0.1, the instance of Overlay for the Polygon data type:

```
instance Overlay TePolygon where ...
```

Example:

```
pol1 = TePolygon [ (Line2d [(1,1),(1,3),(3,3),(3,1),(1,1)]) ]
pol2 = TePolygon [ (Line2d [(2,2),(2,4),(4,4),(4,2),(2,2)]) ]
ps = union [pol1] [pol2]
⇒ [(TePolygon [TeLine2D [(1.0,1.0),(1.0,3.0),(2.0,3.0),
(2.0,4.0), (4.0,4.0),(4.0, 2.0),( 3.0, 2.0),(3.0, 1.0),
(1.0, 1.0) ] ] ] ]
```

Figure 3.15 - Example of overlay operation

3.4.4 Metric operations

TerraHS proves some important metric operations:

- `distance` : calculate the Euclidian distance between two points.
`distance :: TePoint → TePoint → Double`
- `llength`: Returns the length of a Line 2D.
`llength :: TeLine2D → Double`
- `polarea`: Calculates the area of a polygon
`pol_area :: TePolygon → Double`

Examples:

```
l = (TeLine2D [ (1.0,1.0),(1.0,2.0),(1,7) ] )
len = llength l
⇒ 6
dis = distance (TePoint (2,3)) (TePoint (7,3))
⇒ 5
```

Figure 3.16 – Example of metric operations

3.5 Spatial 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 e 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 3.17.

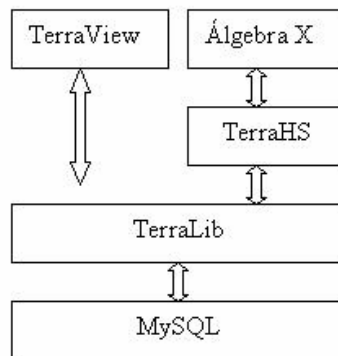


Figure 3.17 - Using the TerraLib to share a geographical database, adapted from Vinhas e Ferreira (2005).

A TerraLib database access does not depends on a specific DBMS and uses an abstract class called TeDatabase (Vinhas e Ferreira, 2005), as shown in Figure 3.18:

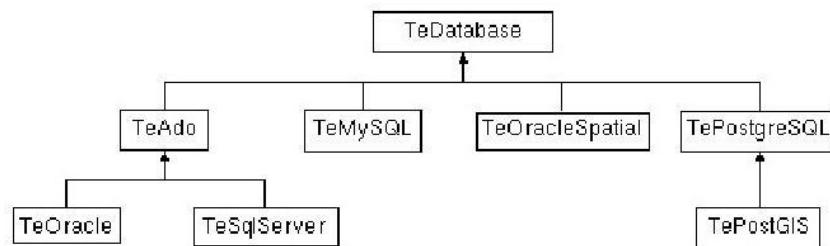


Figure 3.18 - TerraLib database drivers - source: Vinhas and Ferreira (2005)

In TerraHS, the database classes are algebraic data types, where each constructor represents a subclass.

```
data TeDatabase = TeMySQL String String String String
                | TePostgreSQL 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 e Ferreira, 2005).

```
type TeLayerName = String
```

The `Algebras.Base.GeoDatabases` module provides the *GeoDatabases type class*. This *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) → TeLayerName → IO [GeObject]
  store :: (Ptr a) → TeLayerName → [GeObject] → IO Bool
  errorMessage :: (Ptr a) → IO String
```

These operations will then be instantiated to a specific database, such as MySQL.

```
instance GeoDatabases TeDatabase where ...
```

Figure - 3.19 shows an example of a TerraLib database access program.

```
import Algebras.Base.GeoDatabases -- database operations
import TerraHS.TerraLib -- TeMySQL type

host = "sputnik"
user = "Sergio"
password = "terrahs"
dbname = "Amazonia"
main :: IO()
main = do
  -- accessing TerraLib database
  db <- open (TeMySQL 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 - 3.19 - Accessing a TerraLib database using TerraHS

In this chapter we have presented TerraHS, a software developed in Haskell language for GIS application developing. Its main contribution is to provided basic spatial operations and structures for prototyping novel ideas in GisScience. As a validation, we will present in the next chapter the map algebra proposed in Câmara (2005).

CHAPTER 4

A GENERALIZED MAP ALGEBRA IN TERRAHS

4.1 Introduction

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 continuous variables or categorical classifications 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. For this example, we use the map algebra proposed in Câmara et al. (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 the algebra and implement it. We have included the discussion from Câmara et al. (2005) in sections 4.2 and 4.3 as a support for the reader.

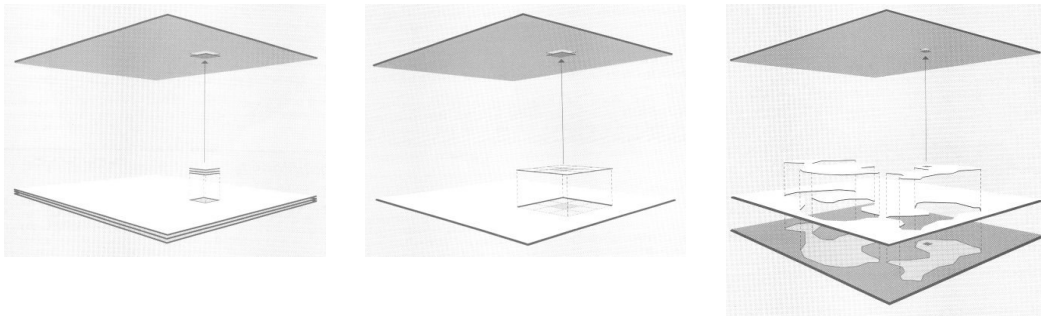
4.2 Tomlin's Map Algebra: a brief review

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. A map is composed by zones, where a zone can contain one or more locations. Tomlin defines three types of higher-order functions for maps. These functions apply a first-order function to all elements of map, according to different spatial restrictions:

- **Local functions.** The value of a location in the output map is computed 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%*" (Figure 4.1 - a)
- **Focal functions.** 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*” (Figure 4.1.b). Focal functions use the condition of adjacency, which matches the spatial predicate *touch*.

- **Zonal functions:** The value of a location in the output map is computed from the values of a spatial neighborhood of the same location in an input map. This neighborhood is a restriction on a second input map. They include expressions such as “*given a map of cities and a digital terrain model, calculate the mean altitude for each city*” (Figure 4.1.c). Zonal functions use the condition of topological containment, which matches the spatial predicate *inside*.



a. Local operation

b. Focal operation

c. Zonal operation

Figure 4.1. Tomlin’s operations for map algebra (source: Tomlin (1983))

There are two classes of functions in map algebra. *First order* functions take values as arguments. *Higher order functions* are functions that have other functions as arguments. Higher order functions are the basis for map algebra operations (Frank, 1997). An example of a higher order function is “*classify as high risk all areas without vegetation with slope greater than 15%*”. In this case, the first-order function is a selection procedure (*test if slope > 15%*) and the higher-order function is the classification function, which applies the selection function to all regions of the map.

Examples of first-order functions include:

- *Single argument mathematical functions:* log, exp, sin, cosine, tan, arcsin, arccosine, arctan, sinh, cosineh, tanh, arcsinh, arccosineh, arctanh, sqrt, power, mod, ceiling, floor.
- *Single argument logical function:* not.
- *Multiargument functions:* sum, product, and, or, maximum, minimum, mean, median, variety, majority, minority, ranking, count.

4.3 Research challenges for map algebra

Tomlin's map algebra has become as a standard way of processing coverages, especially for multicriteria analysis. In recent years, several extensions to map algebra have been proposed. These include the GeoAlgebra of Takeyama and Couclelis (1997), an extension of map algebra that allows for flexible definitions of neighborhoods. Pullar (2001) developed MapScript, a language that allows control structures and dynamical models to be incorporated into map algebra. Ostlander (2004) suggests how map algebra could be embedded in a web service. Mennis et al. (2005) propose an extension of map algebra for spatio-temporal data handling. Frank (2005) discusses how map algebra can be formalized in a functional programming context and how this approach provides support both for spatial and spatio-temporal operations. Nevertheless, all extensions share the *ad hoc* nature of Tomlin's original proposal. They accept the foundations of Tomlin's algebra as a basis for their work.

Therefore, one of the open challenges in spatial information science is to develop a theoretical foundation for map algebra. We need to find out if Tomlin's map algebra can be part of a more general set of operations on coverages. We state these questions as: "*What is the theoretical foundation for map algebra?*" "*Could this theoretical foundation provide support for a more generic map algebra?*"

The proposal by Câmara et al. (2005) is a map algebra that generalizes Tomlin's map algebra by incorporating spatial predicates. The idea is further developed in the next sections and then applied in a functional programming context.

4.4 Spatial predicates as a basis for Map Algebra

As we show in the previous section, spatial operations in Tomlin’s map algebra use only two topological predicates (*touch* and *inside*). It is natural to extend map algebra to use a more general set of spatial predicates. We take the standard set of topological predicates $\{‘disjoint’, ‘equal’, ‘touch’, ‘inside’, ‘overlap’, ‘contains’, ‘intersects’\}$, which cover all vector area-area relations, as proposed by Egenhofer and Herring (1991) and adopted by the Open Geospatial consortium (Ogc, 1996). The 9-intersection model also distinguishes 33 relations between simple lines, 19 between simple lines and simple regions, 2 between points and 3 between points and regions or lines. The works of Winter (1995) and Winter and Frank (2000) extend this definition to the application to raster representations.

The proposal by Câmara et al. (2005) is to develop a map algebra that uses the Open GIS topological spatial predicates. This extended algebra conveys all Tomlin’s algebra operations and enables operations that are not directly expressible by his proposal. In what follows, we show how this extended map algebra can be expressed succinctly and elegantly in Haskell.

4.5 The Open GIS Coverage in Haskell

Our map algebra is based on the *coverage* defined by the Open GIS consortium (Ogc, 2000). A coverage in a planar-enforced spatial representation that covers a geographical area and divides it in spatial partitions that may be either regular or irregular. A coverage is a function $cov :: E \rightarrow A$, where:

- The domain is finite collection, where each element is located in space.
- The range is a set of attribute values.

For each geographic element $e \in E$, a coverage function returns a value $cov(e) = a$, where $a \in A$. A geographical element can represent a location, area, line or point. For retrieving data from a coverage, the Open GIS specification propose describes a discrete function (*DiscreteC_Function*), as shown in Figure 4. below.

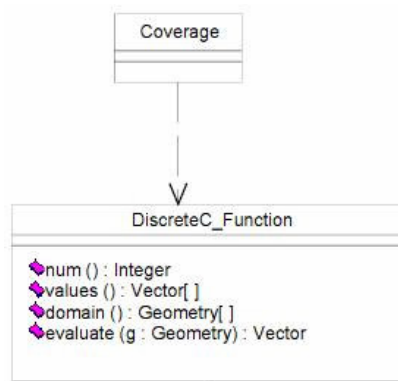


Figure 4.2 - The Open GIS discrete coverage function – source: (Ogc, 2000).

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 define the *type class* Coverages in Haskell.

```

class Coverages cov where
    evaluate :: (Eq a, Eq b) => cov a b → a → Maybe b
    domain   :: cov a b → [a]
    num      :: cov a b → Int
    values   :: cov a b → [b]
    new_cov  :: [a] → (a → b) → (cov a b)
    fun      :: (cov a b) → (a → b)
  
```

The type class *Coverages* 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*:

- *evaluate* is a function that takes a *coverage* and an input value *a* and produces an output value (“give me the value of the coverage at location *a*”).
- *domain* is a function that takes a *coverage* and returns the values of its domain.
- *num* returns the number of elements of the *coverage*’s domain.
- *values* returns the values of the *coverage*’s range.

We propose two extra functions: *new_cov* and *fun*, as described below.

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

We define the *Coverage* data type to use the functions of the generic type class *Coverages*. The *Coverage* data type is also parameterized.

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

The data type *Coverage* has two components:

- 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 *Coverages* to the *Coverage* data type is shown below:

```
instance Coverages Coverage where
  new_cov a f = (Coverage (f, a))
  evaluate f o
    | (elem o (domain f)) = Just ((fun f) o)
    | otherwise = Nothing
  domain (Coverage (f, a)) = a
  num f = length (domain f)
  values f = map (fun f) (domain f)
  fun (Coverage (f, _)) = f
```

Figure 4. show an example of the *Coverage* data type.

```
cl :: Coverage String Int
cl = new_cov ["ab","abc","a"] length
values cl
⇒ [2,3,1]
evaluate cl "ab"
⇒ Just 2
evaluate cl "ad" -- cl does not contain "ad"
⇒ Nothing
```

Figure 4.3 - Example of use of the *Coverage* data type.

4.6 Map Algebra Operations

There are two classes of 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*”.

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). Spatial operations are higher-order functions that use a spatial predicate (some examples of spatial predicates are shown in Figure 4.4). 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).

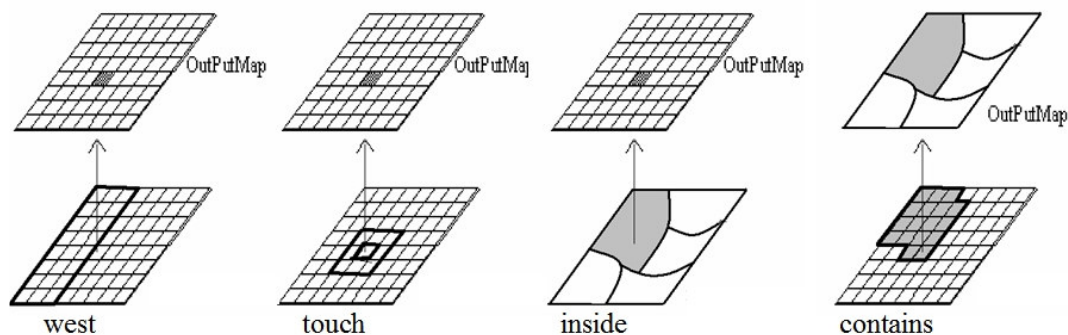


Figure 4.4 Examples of spatial predicates

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 4.1). 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.

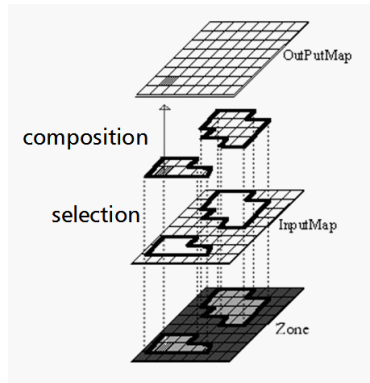


Figure 4.1-Spatial operations (selection + composition).Adapted from (Tomlin, 1990)

4.7 Map Algebra Operations in TerraHS

In TerraHS, we use a generic type class for map algebra operations, as follows:

```
class (Coverages cov) => CoverageOps cov where
  single    :: (b → c) → (cov a b) → (cov a c)
  multiple  :: ([b]→c) → [(cov a b)] → (cov a b)→(cov a c)
  select    :: (cov a b) → (a → c → Bool) → c → (cov a b)
  compose   :: ([b] → b) → (cov a b) → b
  spatial   :: ([b] → b) → (cov a b) → (a → c → Bool)
              → (cov c b) → (cov c b)
```

The implicit assumption of these is that the geographical area of the *output map* is the same as *reference map*.

The instantiation of the coverage operations is provided by:

```
instance CoverageOps Coverage where
  -- non-spatial operation on a single coverages
  single g cl = new_cov (domain cl) ( g . (fun cl))
  -- non-spatial operation on multiple coverages
  multiple fn clist cl = new_cov (domain cl)
    (\x -> fn (faux clist x))
  -- spatial selection
  select cov pr o = new_cov sel_dom (fun cov)
    where sel_dom = [1 | 1 <- (domain cov) , (pr 1 o)]
  -- spatial composition
  compose f cov = (f (values cov))
  -- spatial operation : selection + composition
  spatial fn cl predic cref = new_cov (domain cref)
    (\x -> compose fn (select cl predic x))
```

The *single* function has two arguments: a *coverage* and a first-order function *f*. It returns a new *coverage*, whose domain is the same as the input coverage. The *coverage function* of the output map is the composition of the *coverage function* of the input map and the first-order function *g*. Figure 4. shows an example of a single argument function.

```
single g cl = new_cov (domain cl) ( g . (fun cl))
```

<pre>values cl ⇒ [2, 4, 12] c2 = single square cl values c2 ⇒ [4, 16, 144]</pre>
--

Figure 4.6 - Example of use of the single argument function

The *multiple* function has three arguments: a multivalued function, a coverage list, and a reference *coverage*. It applies a multivalued function to the coverage list. The result has the same domain as the reference coverage. The new coverage function is defined using an auxiliary function that scans the input list:

```
multiple fn clist cl = new_cov (domain cl)
  (\x -> fn (faux clist x))
```

For each location x of the reference coverage, the auxiliary function *faux* applies the multiargument function in the input list of maps. The result is the output value for location x . The function *faux* also handles cases where there are multiargument function fails to returns an output value.

```
faux :: (Eq a, Eq b, Coverages cov) => [(cov a b)] -> a -> [b]
faux [] _ = []
faux (m:ms) e = fauxl (evaluate m e)
    where
        fauxl (Just v) = v : (faux ms e)
        fauxl (Nothing) = (faux ms e)
```

Figure 4. shows an example of *multiple*.

```
values c1
[2, 4, 8]
values c2
=> [4, 5, 10]
c3 = multiple sum [c1, c2] c1
values c3
=> [6, 9, 18]
```

Figure 4.7 - Example of use of *multiple*

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

```
select c prd o = new_map sel_dom (fun c)
    where
        sel_dom = [loc | loc ← (domain m) , (prd loc o)]
```

This function takes a reference element and an input coverage. It creates a coverage that contains all elements of the input that satisfy the predicate over the reference element. Figure 4. shows an example.

```
line= TeLine2D [(1,2),(2,2),(1,3),(0,4)]
domain c1
=> [TePoint(4,5),TePoint (1,2),TePoint (2,3),TePoint (1,3)]
c2 = select c1 intersects line
domain c2
=> [TePoint (1,2), TePoint (1,3)]
```

Figure 4.8 - Example of *select*.

The *composition function* combines selected values using a multivalued function. In Figure 4., the `compose` function is applied to *coverage c1* and to the multivalued function *sum*.

```
compose f m = (f (values m))
```

```
values c1
⇒ [2, 6, 8]
compose sum c1
⇒ 16
```

Figure 4.9 - Example of *compose*.

The `spatial` function combines spatial selection and composition. The output coverage has the same domain as the reference coverage. For each location in the output coverage, the *selection function* produces a set of values that satisfy a spatial predicate.. The *composition function* uses the selected values to produce the result. Figure 4.10 shows an example.

```
spatial fn c prd cref = new_cov (domain cref)
                               (\x → compose fn (select c prd x))
```

```
domain c1
= [TePoint(4,5),TePoint (1,2),TePoint (2,3),TePoint (1,3)]
values c1
= [2,4,5,10]
domain c2
= [(TeLine2D[(1,2),(2,2),(1,3),(0,4)])]
c3 = spatial sum c1 intersects c2
values m3 -- 4 + 10
= [14]
```

Figure 4.10 - Example of *spatial*

The `spatial` operation selects all points of *c1* that intersect *c2* (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.

4.8 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.

4.8.1 Storage and Retrieval

Since a *Coverage* 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.

```
toCoverage:: [GeObject]→ AttrName→ (Coverage Geometry Value)
toGeObject:: (Coverage Geometry Value)→ AttrName→ [GeObject]
```

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

```
retrieveCov::
  TeDatabase→ LayerAttr→ IO (Coverage Geometry Value)
retrieveCov db (layername, attrname) = do
  db <- open db
  geoset <- retrieve db layername
  let map = toCoverage geoset attrname
  close db
  return map
```

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

```

storeCov:: TeDatabase→ LayerAttr
          → (Coverage Geometry Value) → IO Bool
storeCov db (layername, attrname) c = do
  let geos = toGeObject c attrname
  db <- open db
  close db
  let status = store db layername geos
  return status

```

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

```

host = "sputnik"
user = "Sergio"
password = "terrahs"
dbname = "Amazonia"
main:: IO ()
main = do
  let db = (TeMySQL host user pass dbname)
  cov <- retrieveCov db ("amazonia","deforest")
  -- apply a nonspatial operation
  let defclass = single classify cov
  storeCov db ("amazonia", "defclass") defclass

```

Figure 4.11 - Retrieving and storing a *coverage* in a TerraLib database

4.8.2 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, from (Aguiar, 2006), 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 4.2).

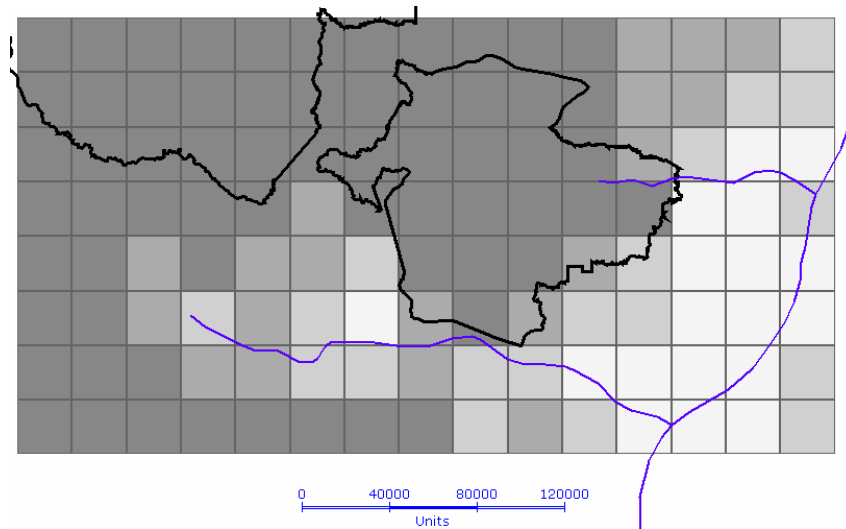


Figure 4.2 – Deforestation, Protection Areas and Roads Maps (Pará State)

Our first example considers the expression: “*Given a coverage 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 coverage using the `single` operation together with the `classify` function:

```
def_class = single classify def_cov
```

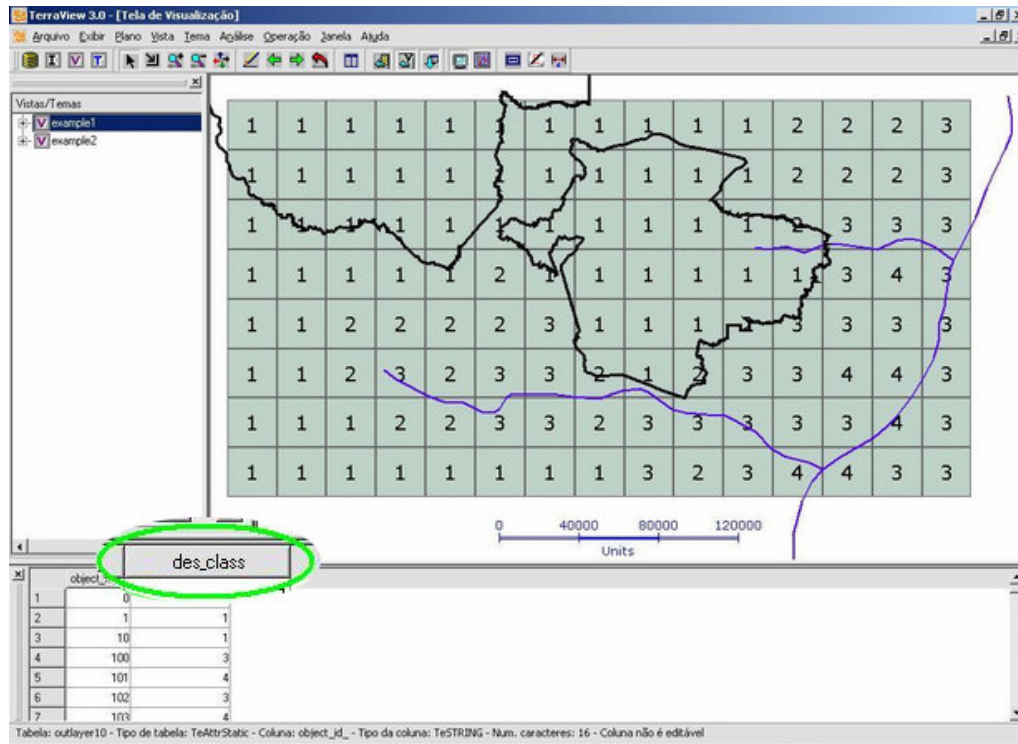


Figure 4.3 – The classified coverage

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

`def_prot = spatial mean def_cov within prot_areas`

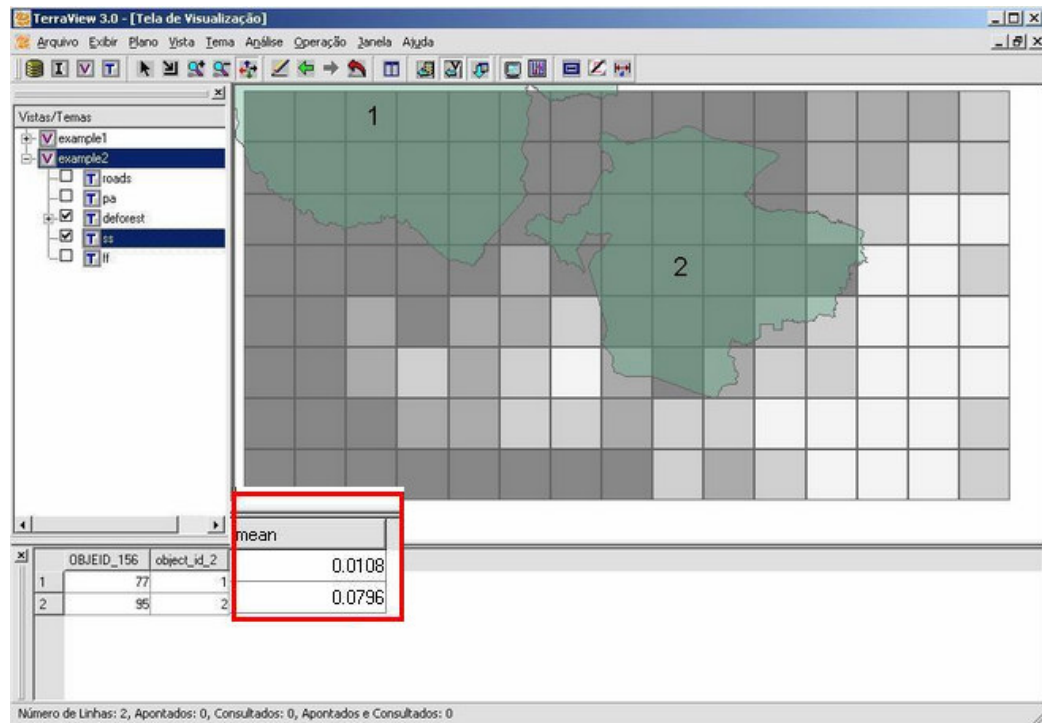


Figure 4.4 – Deforestation mean by protection area

In our third example, we consider the expression: “Given a coverage containing roads and a deforestation coverage, calculate the mean of the deforestation along the roads”. We have as inputs: the deforestation coverage (def_cov), a spatial predicate (intersect), a multivalued function (mean) and a road map (roads). The product is a coverage with one value for each road. This value is the mean of the cells that intercept this road.

```
road_def = spatial mean def_cov intersect roads
```

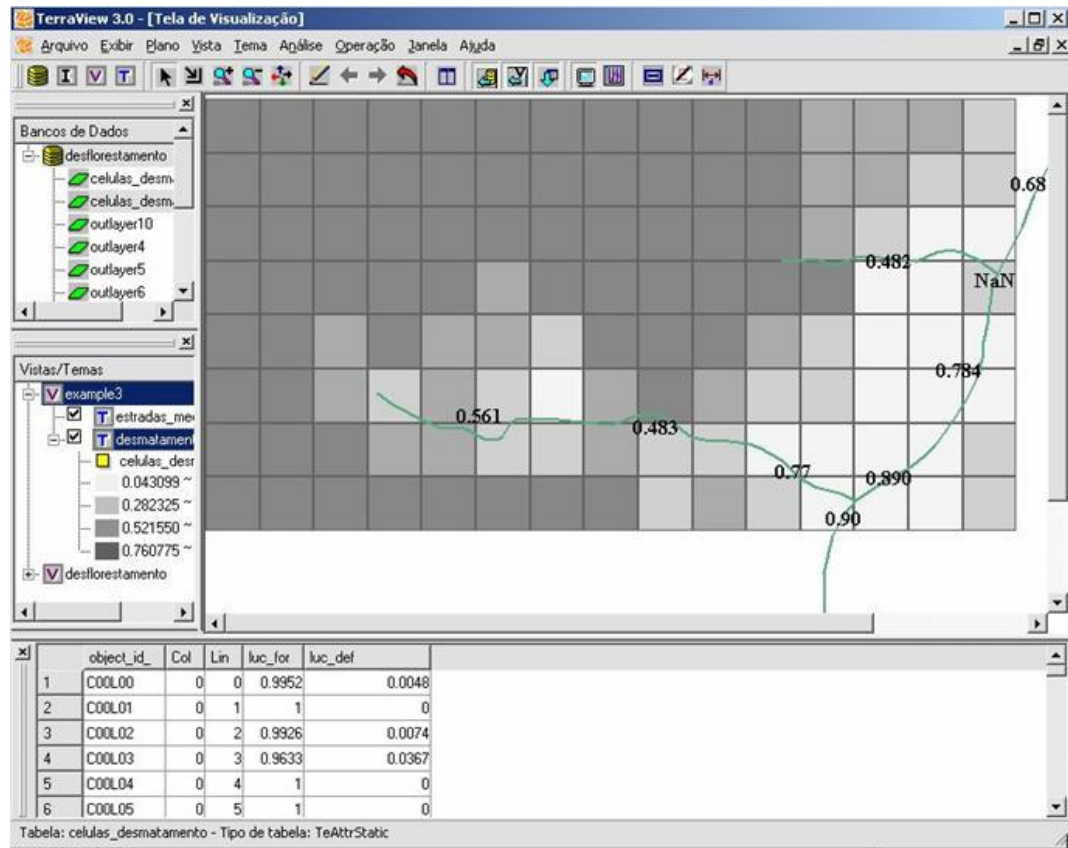


Figure 4.5 – Deforestation mean along the roads

4.9 Discussion of the Results

The implementation of a map algebra in TerraHS shows the benefits of using functional programming for GIS application development. The resulting map algebra is compact, generic and extensible. It is also useful to compare this implementation with a similar product in an imperative language. Table 4.1 presents the total number of Haskell lines used to develop the map algebra.

Table 4.1 – 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. The example shows the benefits on using functional programming, since it enables a fast prototyping and testing cycle.

CHAPTER 5

CONCLUSION AND FUTURE WORKS

The hypothesis of our work was: *to integrate functional programming and spatial databases for GIS application development, we should build a functional GIS on top of an existing spatial database support.* We then use each programming paradigm in the most efficient fashion. To assess our hypothesis, we have built **TerraHS**, an application development system that enables developing geographical applications in the Haskell functional language. **TerraHS** uses the data handling facilities of the C++ GIS library TerraLib. We then applied TerraHS to the concrete case of developing a map algebra. We have shown that the resulting product is more expressive, more modular, and was developed faster than its equivalent in C++.

Combining **TerraHS** and **TerraLib** enables the use of functional programming to real-life problems, and is a contribution to make Haskell a more widely used tool for GIS application development. We use each programming style in the most efficient fashion. We rely on imperative languages such as C++ to provide spatial database support and we use functional programming for building parts that provide data manipulation algorithms.

We now consider some future works:

- **Database Access:** The current version of TerraHS has a core of database access operations. It can write a new record but not change an existing one. Future versions of TerraHS will include operations such as *update*.
- **Spatiotemporal algebras:** TerraHS is a good environment for development spatiotemporal algebras. For example, Güting (2005) defines an algebra for moving objects. His spatio-temporal data types for moving objects are embedded in a query language to answer queries as: “*Given the trajectories of two airplanes, when they will pass over the same location?*”. Similarly, Medak

(2001) proposes an algebra for modeling change in socio-economical units. Medak's algebra provides answers to queries such as: "*When was this parcel divided?*" By using TerraHS, we can envisage developing a set of data types and operators for handling all types of spatiotemporal data.

- **Integration into GUI software:** To improve the impact of applications developed in TerraHS, we need to integrate it into a user-friendly environment. This integration needs to provide an external shell that hides the more hard to learn parts of Haskell.

We now summarize our main findings and contributions. The Haskell language provides efficient support for using functional programming for real applications. Haskell enforces a programming style that is both rigorous and expressive. Most GIS applications contain a core part that can be expressed by algebraic data types and thus can be nicely developed in Haskell. Applications such as TerraHS provide the missing link between spatial and spatiotemporal algebras and spatial databases. Our work has thus validated our initial hypothesis and provided a software component that can be useful in practice. We hope that TerraHS will be useful for developing complex and sound GIS applications using an innovative programming style.

REFERENCES

- AGUIAR, A. P. D. D. **Modeling Land Use Change in the Brazilian Amazon: Exploring Intra-Regional Heterogeneity**. (Doctor Thesis). Instituto Nacional de Pesquisas Espaciais (INPE), São José dos Campos, 2006.
- BACKUS, J. Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs. **Communications of the ACM**, v.21, n.8, p.613-641. 1978.
- CÂMARA, G. Representação Computacional De Dados Geográficos. In: M. Casanova, G. Câmara, *et al* (Ed.). **Bancos De Dados Geográficos**. Curitiba: MundoGeo Editora, 2005. Representação Computacional De Dados Geográficos, p.11-52
- CÂMARA, G., et al. Towards a Generalized Map Algebra: Principles and Data Types. In: **VII Workshop Brasileiro de Geoinformática**. Campos do Jordão: SBC, 2005. p.
- CÂMARA, G., et al. Spring: Integrating Remote Sensing and Gis with Object-Oriented Data Modelling. **Computers and Graphics**, v.15, n.6, p.13-22. 1996.
- CASANOVA, M., et al., Eds. Bancos De Dados Geograficos (Spatial Databases). Curitiba: Editora MundoGEOed. 2005.
- CHAKRAVARTY, A. P. A. M. **Interfacing Haskell with Object-Oriented Language**. In: **15th International Workshop on the Implementation of Functional Languages**. Lübeck, Germany: Springer-Verlag, 2004. p.
- CHAKRAVARTY, M. The Haskell 98 Foreign Function Interface 1.0: An Addendum to the Haskell 98 Report. 2003.
- CHAKRAVARTY, M. M. T. C->Haskell, an Interface Generator for Haskell. 2005.
- DAUME, H. Yet Another Haskell Tutorial 2004.
- EGENHOFER, M. e HERRING, J. **Categorizing Binary Topological Relationships between Regions, Lines, and Points in Geographic Databases**. Orono, ME: Department of Surveying Engineering, University of Maine. 1991
- ERWIG, M., et al. Spatio-Temporal Data Types: An Approach to Modeling and Querying Moving Objects in Databases. **GeoInformatica**, v.3, n.3, p.269-296. 1999.
- FINNE, S., et al. Calling Hell from Heaven and Heaven from Hell. Proceedings of the fourth ACM SIGPLAN international conference on Functional programming. Paris, France: ACM Pressersity Press: 114-125 p. 1999.

FRANK, A. Higher Order Functions Necessary for Spatial Theory Development. In: **Auto-Carto 13**. Seattle, WA: ACSM/ASPRS, 1997. 11-22 p.

FRANK, A. One Step up the Abstraction Ladder: Combining Algebras - from Functional Pieces to a Whole. In: **COSIT - Conference on Spatial Information Theory**: Springer-Verlag, 1999. 95-108 p.

FRANK, A. Map Algebra Extended with Functors for Temporal Data. In: **Conceptual Modeling: ER 2005 Workshops**. Klagenfurt, Austria,: Springer, 2005. p.

FRANK, A. e KUHN, W. Specifying Open Gis with Functional Languages. In: M. Egenhofer e J. Herring (Ed.). **Advances in Spatial Databases—4th International Symposium, Ssd '95, Portland, Me**. Berlin: Springer-Verlag, v.951, 1995. Specifying Open Gis with Functional Languages, p.184-195. (Lecture Notes in Computer Science)

FRANK, A. e MEDAK, D. Executable Axiomatic Specification Using Functional Language - Case Study: Ontology for a Spatio-Temporal Database. 1997.

GÜTING, R., et al. Implementation of the Rose Algebra: Efficient Algorithms for Realm-Based Spatial Data Types. In: M. Egenhofer e J. Herring (Ed.). **Advances in Spatial Databases—4th International Symposium, Ssd '95, Portland, Me**. Berlin: Springer-Verlag, v.951, 1995. Implementation of the Rose Algebra: Efficient Algorithms for Realm-Based Spatial Data Types, p.216-239. (Lecture Notes in Computer Science)

GÜTING, R. H., et al. Spatio-Temporal Models and Languages: An Approach Based on Data Types. In: M. Koubarakis (Ed.). **Spatio-Temporal Databases**. Berlin: Springer, 2003. Spatio-Temporal Models and Languages: An Approach Based on Data Types

GÜTING, R. H. e SCHNEIDER, M. **Moving Objects Databases**: Morgan Kaufmann Publishers. 2005

GUTTAG, J. e HORNING, J. The Algebraic Specification of Abstract Data Types. **Acta Informatica**, v.10, p.27-52. 1978.

HUDAK, P. Conception, Evolution, and Application of Functional Programming Languages. **ACM Comput. Surv.**, v.21, n.3, p.359-411. 1989.

HUDAK, P., et al. A Gentle Introduction to Haskell 98 1999.

HUDAK, P., et al. The History of Haskell (Draft) 2007.

HUGHES, J. Why Functional Programming Matters. **Computer Journal**, v.32, n.2, p.98--107. 1989.

JONES, M. P. Functional Programming with Overloading and Higher-Order Polymorphism. Advanced Functional Programming, First International Spring School

on Advanced Functional Programming Techniques-Tutorial Text %@ 3-540-59451-5: Springer-Verlag: 97-136 p. 1995.

JONES, S. L. P. e WADLER, P. Imperative Functional Programming. In: **Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages**. Charleston, South Carolina, United States: ACM Press, 1993. 71-84 p.

LIN, F.-T. Many Sorted Algebraic Data Models for Gis. **International Journal of Geographical Information Science**, v.12, n.8, p.765-788. 1998.

MCCARTHY, J. A Basis for a Mathematical Theory of Computation. In: **Computer Programming and Formal Systems**: North-Holland, Amsterdam, 1963. 33--70 p.

MEDAK, D. **Lifestyles - a New Paradigm in Spatio-Temporal Databases**. Department for Geoinformation, Technical University of Vienna, Vienna, 1999.

MEDAK, D. Lifestyles. In: A. U. Frank, Raper, J., & Cheylan, J.-P. (Ed.). **Life and Motion of Socio-Economic Units. Esf Series**. London: Taylor & Francis, 2001. Lifestyles

MEIJER, E. e FINNE, S. Lambada, Haskell as a Better Java. In: **Proc. Haskell Workshop 2000**, 2000. p.

MENNIS, J., et al. Cubic Map Algebra Functions for Spatio-Temporal Analysis. **Cartography and Geographic Information Science**, v.32, n.1, p.17-32. 2005.

MEYER, B. On to Components. **Computer** v.32, n.1, p.139-140. 1999.

MOGGI, E. Notions of Computation and Monads. **Inf. Comput.**, v.93, n.1, p.55-92. 1991.

OGC, Ed. The Opengis[®] Guide - Introduction to Interoperable Geoprocessing and the Opengis Specification. Boston: Open GIS Consortium, Inc. 1996.

OGC. Open Gis Consortium. Topic 6: The Coverage Type and Its Subtypes. 2006 2000.

OSTLÄNDER, N. Interoperable Services for Web-Based Spatial Decision Support. In: **7th AGILE Conference on Geographic Information Science**. Heraklion, Greece: AGILE, 2004. p.

PEYTON JONES, S. Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-Language Calls in Haskell. In: **Engineering theories of software construction**: IOS Press, 2001. 47-96 p.

PEYTON JONES, S. Haskell 98 Language and Libraries the Revised Report 2002.

PEYTON JONES, S., et al. Haskell 98: A Non-Strict, Purely Functional Language 1999.

PEYTON JONES, S., et al. Scripting Com Components in Haskell. In: **Proceedings of the 5th International Conference on Software Reuse** IEEE Computer Society, 1998. 224 p.

PIERCE, B. **Types and Programming Languages**. Cambridge, MA: MIT Press. 2002

PULLAR, D. Mapscript: A Map Algebra Programming Language Incorporating Neighborhood Analysis. **GeoInformatica**, v.5, n.2, p.145-163. 2001.

SHIELDS, M. e JONES, S. L. P. Object-Oriented Style Overloading for Haskell. **Electronic Notes in Theoretical Computer Science**, v.59, n.1. 2001.

TAKEYAMA, M. e COUCLELIS, H. Map Dynamics: Integrating Cellular Automata and Gis through Geo-Algebra. **International Journal of Geographical Information Systems**, v.11, n.1, p.73-91. 1997.

THOMPSON, S. **Haskell: The Craft of Functional Programming**. Harlow, England: Pearson Education. 1999

TOMLIN, C. D. A Map Algebra. In: (Ed.). **Harvard Computer Graphics Conference**. Cambridge, MA, 1983. A Map Algebra

TOMLIN, C. D. **Geographic Information Systems and Cartographic Modeling**. Englewood Cliffs, NJ: Prentice-Hall. 1990

VINHAS, L. e FERREIRA, K. R. Descrição Da Terralib. In: M. Casanova, G. Câmara, *et al* (Ed.). **Bancos De Dados Geográficos**. Curitiba: MundoGeo Editora, 2005. Descrição Da Terralib, p.397-439

WADLER, P. Comprehending Monads. In: **Proceedings of the 1990 ACM conference on LISP and functional programming** Nice, France: ACM Press, 1990. 61-78 p.

WINTER, S. Topological Relations between Discrete Regions. In: M. Egenhofer e J. Herring (Ed.). **Advances in Spatial Databases—4th International Symposium, Ssd '95, Portland, Me**. Berlin: Springer-Verlag, v.951, 1995. Topological Relations between Discrete Regions, p.310-327. (Lecture Notes in Computer Science)

WINTER, S. e FRANK, A. Topology in Raster and Vector Representation. **GeoInformatica**, v.4, n.1, p.35-65. 2000.

WINTER, S. e NITTEL, S. Formal Information Modelling for Standardisation in the Spatial Domain. **International Journal of Geographical Information Science**, v.17, p.721--741. 2003.

YAKELEY, A. Haskell: Libraries and Tools. 2006 2006.