Frank, Andrew U., Silvia Nittel, and Stephan Winter. "Abstract Modeling with Functional Languages." 5. Vienna: Department of Geoinformation, TU Vienna, 2000.

OPENGIS PROJECT DOCUMENT 00-030

TITLE:		**Abstract Modeling with Functional Languages**
AUTHOR:	**Name:**	**Andrew U. Frank, Silvia Nittel, Stephan Winter**
	Address:	**TU Vienna and UCLA**
	Phone:	**+43 1 58801-12700**
	FAX:	**+43 1 58801-12799**
	Email:	**{frank,winter}@geoinfo.tuwien.ac.at**
DATE:		**May 8, 2000**
CATEGORY:		**Information**

1. Background

Recently a discussion came up on the TC email reflector about the role of the Abstract Specification (OGC 1999) in the specification process of OGC. The background of this discussion is dissatisfaction with the current state of the abstract models in the specs, with regard to consistency, to its actuality, and to its value compared to rapid prototyping. We deem the abstract model necessary, but we admit the complexity of creating and maintaining this model. At this point we present a specification technique, namely algebraic specification by second order (functional) languages, that has some advantages compared to the tools in use in OGC.

2. References

OGC, 1999: The OpenGIS Abstract Specification - Topic 0: Abstract Specification Overview. 99-100r1, http://www.opengis.org/public/abstract/99-100r1.pdf.

3. Proposal

3.1 Introduction

The language used for the description of the specifications is a very important tool for OGC, which directly influences the quality of the specifications produced. Currently UML (Booch et al. 1999; OMG 2000) is used, which is quite flexible, but does not fulfill all requirements for a specification language. UML was developed as a language to document software systems and help with the development of software. It was not primarily designed for specification writing.

In the specifications as currently written, some situations occur regularly, which could be described with a more powerful language more succinctly. Second order languages, i.e. languages, which allow functions as variables, are more flexible without being more complex or lacking in mathematical rigor. We use here the syntax of a concrete, practically tested functional language, Haskell (Hudak et al. 1999; Jones and Peterson 1999), which can be downloaded from http://www.haskell.org.

The discussion paper describes first two often-occurring situations and shows how they are more clearly expressed with second order languages. In a second section we present some other advantageous properties of specifications in a functional language. We then discuss, how in advanced functional languages the distinction of abstract spec and implementation spec can be expressed clearly with a clean linkage between the two.

3.2 Two common topoi in specifications

3.2.1 Composition of functions

The combination of functions, such that one is applied to the result of the previous one, is often necessary. For example in coordinate transformations, the translation and the rotation are combined one after the other. This is mathematically written as

```
translate (v1, rotate (alpha, v2))
```

Functional languages have a primitive operation for function composition, written as "." and the combined transformation (without argument) is written as

```
(translate v1) . (rotate alpha)
```

One notes the different use of parenthesis – strictly to group, but not to separate the arguments from a function name. The result of function composition is a function

```
transform v1 alpha = translate v1 . rotate alpha
```

This function `transform` can be applied to any vector `v`

```
vt = transform v1 alpha v
```

3.2.2 Mapping of operations over collections

Often an operation is defined as applicable to a single object – e.g., the transformation above applies to a single vector –, and then the same operation must be applied to all elements in a collection. A second order function `map` is usually predefined to 'map' the operation to all elements.

To transform all points in a collection `c1` of points, we can write

```
ct = map (transform v1 alpha) c1
```

We even can define a function to transform all elements in a collection:

```
transformList v1 alpha x = map (transform v1 alpha) x
```

This saves the complication that operations applicable to single instances and a second operation applicable to collections must be written in the abstract specs. Implementation specs for languages where two different names for such operations are necessary can contain names for both.

3.3 Other advantages of functional languages

Due to limited space here we present only two other advantages of using functional languages for abstract specifications. These are possibility to compose several algebras, and to execute the code.

3.3.1 Composition of algebras

OGC is developing its abstract specification in a series of books (topics 1 to 16, actually). With the presented tool, one creates a many-sorted algebraic specification for a topic. Let us consider the Coverage (topic 6): the central types to be specified are the `Coverage`, the `C_Function`, and their derivatives. However, the `Coverage` is treated as a sub-type of `Feature`, where `Feature` is covered by topic 5. During the development of the many-sorted algebra of topic 6, only a very coarse concept of `Feature` is required ("A 'feature' has a 'type' and a list of 'properties'"). As long as this coarse concept provides correct

interfaces (corresponding to the topic 5), the coarse specification in topic 6 can be replaced at any time by the complete many-sorted algebra of topic 5. Composition of algebras is possible without conflicts due to the module concept of Haskell and qualified namespaces.

3.3.2 Execution of code

Haskell code is executable (primarily by an interpreter, but compilers are available too). Loading a specification into the interpreter proves (guarantees) consistency of names and types. Adding test cases to the specifications allows 'run-throughs' of the code, observing its behavior.

Example. Consider the method `domain` of a `C_Function` (topic 6). Given any `C_Function`, e.g., a simple grid of size 1x1, the method `domain` can be applied on it and the result can be evaluated:

```
cf :: CFunction
cf = ...
test :: [Geometry]
test = domain cf
```

In our example, the result of executing the `test` function is the domain of the 1x1-grid, which is a single point.

- Testing the behavior of specifications opens new possibilities for the communication between domain experts and abstract modelers. Correctness of the abstract model against the essential model can be checked by typical test cases. Completeness of the abstract model can be checked by critical test cases.
- Implementers can check their implementations against the spec, gaining correct implementations of an implementation spec. Even compliance testing could be done by specifying a reference set of test cases and their compulsory results.

3.4 Separation between Abstract Specification and Implementation Specification

The separation between an abstract specification, which describes the intended behavior independent of an implementation and the implementation spec, which describes a particular implementation of this abstract behavior, is a key feature of the OGC specification process and makes the abstract spec available for systems deployed on multiple platforms. This difference is not well expressed in the current written specifications; the same language elements and forms appear in both abstract spec and implementation spec. For example, a UML-like notation of the `GeometryValuePair` (topic 6) in Haskell would be:

```
class GeometryValuePairs a where
   geom  :: a -> Geometry
   value :: a -> Vector
```

This notation of an abstract model is somehow mixing with implementation aspects using specific data types (`Geometry`, `Vector`). In a modern functional language like Haskell, the abstract description of behavior (abstract spec) and the concrete realization with particular data types can be separated. Abstract specifications are given as descriptions of operations on abstract classes, represented by type parameters:

```
class (Geometries g, Vectors v) => GeometryValuePairs a g v | a -> g v where
   geom  :: a -> g
   value :: a -> v
```

The conditions for the types being specific (e.g., a geometry or a value) is no longer written as a type, but expressed as a condition that the actual type must be an instance of a determined class (e.g., `Geometries` or `Vectors`).

The corresponding implementation spec is written as instances of these classes. Here the type parameters must be replaced by concrete types (the structure of which is given as data definitions). The language processor is checking that the stated conditions are fulfilled.

```
class Geometries g where …
instance Geometries Geometry where …
class Vectors v where …
instance Vectors Vector where …
instance GeometryValuePairs GeometryValuePair Geometry Vector where …
```

Independent of this clear separation in the abstraction level, axioms describing the outcome of the operation (i.e. the semantics or the behavior) can be stated at the abstract or implementation level.

3.5 Conclusion

This discussion paper can present only the most important concepts, and short examples of using functional languages for abstract specifications. The interested reader is referred to a series of research papers transferring algebraic specifications (Guttag and Horning 1978; Liskov and Guttag 1986; Loeckx et al. 1996) to the domain of Geoinformation (Frank and Kuhn 1995; Kuhn 1997; Frank 1999; Frank and Kuhn 1999). Two of the authors are working on a research paper specifying a topic of OGC (the coverage) with Haskell; this paper is available on request (Nittel and Winter in preparation).

Our goal was to demonstrate the advantages of using a functional language for abstract modeling. We have discussed two common topoi of abstract modeling in a functional language, we have shown other properties of algebraic specifications useful in abstract modeling, and finally we have demonstrated the possibility to separate clearly abstract and implementation specification.

3.6 References

Booch, G.; Rumbaugh, J.; Jacobson, I., 1999: The Unified Modeling Language Reference Manual. Addison-Wesley, Reading, 550 pp.

Frank, A.U., 1999: One step up the abstraction ladder: Combining algebras - From functional pieces to a whole. In: Freksa, C., Mark, D. (Eds.), Spatial Information Theory. Lecture Notes in Computer Science 1661, Springer-Verlag, Berlin.

Frank, A.U.; Kuhn, W., 1995: Specifying Open GIS with Functional Languages. In: Egenhofer, M.J.; Herring, J.R. (Eds.), Advances in Spatial Databases. Lecture Notes in Computer Science 951, Springer-Verlag, Berlin, pp. 184-195.

Frank, A.U.; Kuhn, W., 1999: A Specification Language for Interoperable GIS. In: Goodchild, M.F. et al. (Eds.), Interoperating Geographic Information Systems, Kluwer, Norwell, MA, pp. 123-132.

Guttag, J.V.; Horning, J.J., 1978: The Algebraic Specification of Abstract Data Types. Acta Informatica, 10: 27-52.

Hudak, P.; Peterson, J.; Fasel, J.H., 1999: A Gentle Introduction to Haskell 98, http://www.haskell.org/tutorial/.

Jones, M.P.; Peterson, J.C., 1999: Hugs 98 User Manual. (Part of the Hugs distribution). Oregon Graduate Institute and Yale University, http://www.haskell.org/hugs/.

Kuhn, W., 1997: Approaching the Issue of Information Loss in Geographic Data Transfers. Geographical Systems, 4(3): 261-276.

Liskov, B.; Guttag, J., 1986: Abstraction and Specification in Program Development. The MIT Electrical Engineering and Computer Science Series. The MIT Press, Cambridge, 469 pp.

Loeckx, J.; Ehrich, H.-D.; Wolf, M., 1996: Specification of Abstract Data Types. Wiley-Teubner, Chichester, 260 pp.

Nittel, S.; Winter, S., in preparation: Formalisation of Spatial Standards.

OMG, 2000: UML Resource Page, http://www.omg.org/uml/.