Metamodel and UML Profile for Functional Programming Languages

Marcin Szlenk[1]

[1] Warsaw University of Technology, Institute of Control & Computation Engineering, Nowowiejska 15/19, 00-665 Warsaw, Poland, m.szlenk@ia.pw.edu.pl

Abstract. Functional programming languages are ideally suited for developing dependable software, but not much work have been done on modeling functional programs. Although UML is mainly based on concepts which are native to imperative object-oriented programming languages, this chapter shows how – through the profile mechanism – it can be used to model software that is to be implemented in a functional programming language. In this chapter Haskell was chosen as one of the most popular modern, pure functional languages. First, a partial metamodel of Haskell is defined and then the corresponding UML profile is presented.

1 Introduction

Unified Modeling Language (UML) [11, 14] is intended to be a universal general-purpose modeling language for software systems. UML contains an extensibility capability for customizing models for particular domains or platforms, where UML extensions are organized into *profiles*. Basic UML – without profiles – reflects the imperative and object-oriented paradigm. A system is modeled as a collection of discrete objects that interact to perform given work. Using UML to model software that is to be implemented in languages supporting other programming paradigms may seem to be odd at first, however, creating a dedicated UML profile may give a natural and convenient modeling notation, which at the same time benefits from the existing tool support for UML. This chapter is an attempt at defining the UML profile for a programming language built on a functional programming paradigm.

2 Functional Programming

Functional programming treats computations as the evaluation of functions (or expressions), avoiding using state and mutable data [1, 4]. Thus, functions are stated in a declarative way, where in contrast to the imperative programming, a

function definition shows what is to be done, rather than how it is to be done in terms of state changes. Functions are treated here as any other values, that is, they can be passed as arguments to other functions or be returned as a result of a function. Some functional programming languages, e.g. ML variants like Standard ML [8], Objective Caml [13] or F# [3], allow to program in both functional and imperative (including object-oriented) style, while the others, e.g. Miranda [16] or Haskell [5, 6], lack imperative programming constructs and remain *purely* functional.

As far as UML modeling is concerned, one can distinguish two types of models: dynamic and static. The dynamic model is used to express the behaviour of a system over time, whereas the static model shows those aspects that do not change over time. The dynamism is intuitively understood here as changes in a system state (which is constituted of the states of its objects), however, in pure functional programs there is no concept of state or order of execution. It is left up to the runtime system how to compute the values given the relations to be satisfied between them. In that sense, in a functional programming language (or at least in its pure subset) only the static structure of program is explicitly specified, while the dynamic aspects remain hidden.

2.1 Haskell

Haskell is nowadays probably the most popular purely functional programming language. It has been designed as a vehicle for functional programming teaching, research, and applications and efforts in improving it are still ongoing. In this chapter the current Haskell specification [6] is used.

In Fig. 1 a sample program written in Haskell is presented. This is a simplified example taken from [6]. The program is organized into one module called '`AStack`', which contains a user-defined *algebraic data type* '`Stack`' (parameterized with a *type variable* '`a`') and functions '`push`', '`size`', '`pop`', and '`top`' for typical stack operations. The type variable '`a`' can be replaced by concrete types (such as '`Int`', '`Float`', and so on) when a given value of type '`Stack`' is declared or the functions are applied. This serves as a parametric polymorphism mechanism.[1] Both the data type and the functions are explicitly (they appear on an *export list*) exported by the module and are available to anyone importing the module. This sample program will be used later to explain and present the application of a proposed UML profile, however, the basic knowledge of functional programming is assumed.

[1] The other kind of polymorphism called *overloading* can be defined in Haskell using *type classes*.

```
module AStack (Stack, push, pop, top, size) where

data Stack a = Empty | MkStack a (Stack a)

push :: a -> Stack a -> Stack a
push x s = MkStack x s

size :: Stack a -> Int
size s = length (stkToLst s) where
            stkToLst  Empty         = []
            stkToLst (MkStack x s)  = x : stkToLst s

pop :: Stack a -> (a, Stack a)
pop (MkStack x s) = (x, s)

top :: Stack a -> a
top (MkStack x s) = x
```

Fig. 1 A sample Haskell program

3 Metamodel

Although Haskell contains some unique features, this chapter will stick to a subset which is common to several other functional programming languages. In particular, the system of *type classes* [6] will be omitted. It not only simplifies further consideration but also allows to easily adapt the proposed profile to other functional languages.

The strategy of defining the UML profile for Haskell is here similar to the one used in UML profile specifications provided by Object Management Group (OMG), e.g. [10]. First, the Haskell *metamodel* (a model of Haskell expressed in UML) is defined. The goal of defining this metamodel is to set the scope of Haskell language which will be included in the profile and to set the level of abstraction for the profile elements. The metamodel presented below is intended to provide sufficient details to create Haskell design and implementation models.[2] The assumed level of abstraction allows to partially generate Haskell code (that will need to be further completed by hand), but does not allow for full code generation from models. As stated before, it is not a complete metamodel of the Haskell language yet it describes a consistent and useful subset of the language. The UML profile corresponding to this metamodel will be then defined in Sect. 4 Profile.

[2] They correspond to Platform Specific Models (PSM) in Model Driven Architecture (MDA) approach [9].

3.1 Haskell Metamodel

The Haskell metamodel is presented as three class diagrams completed with a description of the important features of the diagram and additional constraints expressed in Object Constraint Language (OCL) [7].

Module contents (Fig. 2) Haskell programs are organized into *modules*, which play a similar role to packages in Java or namespaces in C++ language. Two kinds of basic program elements defined in such a module are *functions* and *user data types*, but it is not obligatory to define them in a certain module, i.e. the '`module Name where`' header at the beginning of a file (see Fig. 1) can be omitted.[3] Before the '`where`' keyword a parenthesized list of functions, types and constructors exported by a module can be added. The attribute '`isExported`' in the metamodel indicates whether the element appears on the export list of a module. Modules can also import other modules (their exported elements) by adding the '`import`' declarations at the beginning of the module.[4] In fact, the module system in Haskell allows also for importing chosen elements of modules and hiding others. Module imports may not form a cycle. Note that no additional constraints in the metamodel are needed here because this fact results from the semantics of the aggregation relationship in UML, which is transitive and antisymmetric [11].

Haskell functions take zero or more arguments and must always return a result. A zero-argument function is called a *value*. Haskell is a statically and strongly typed language, but the user does not have to explicitly specify the types of functions as they can be inferred by the system. Functions in Haskell are pure, i.e. they do not have any *side-effects*. To examine and modify the current state of the world, e.g. read and write files, read from a keyboard or print something on a screen, one has to use IO (input/output) *actions*. Every IO action returns a value, but in the type system the returned value is tagged with '`IO`' type, distinguishing actions from functions. For example, the type of the function '`getChar`' is:

```
getChar :: IO Char,
```

what means that this function is actually an action and when it is invoked, the result will have type '`Char`'. IO actions can be passed to functions. The attribute '`isIO`' in the metamodel indicates whether the type is an '`IO`' type.

User data types are defined using '`data`' keyword (see Fig. 1). They are algebraic types, i.e. any value of such a type is created using a *constructor*, which is just a function, expecting some arguments (of other types) and delivering a value of the given user type. A constructor may also not take any arguments (may be a value) and an algebraic type may have many constructors (these are separated with

[3] In this case, the header is assumed to be '`module Main (main) where`'.

[4] In fact, the module system in Haskell allows also for importing chosen elements of modules and hiding others.

the '|' character). A constructor cannot be an action and its result type is the user type whose values it constructs. This constraint can be expressed in OCL as below:

```
Constructor
resultType = userType and resultType.isIO = False.
```

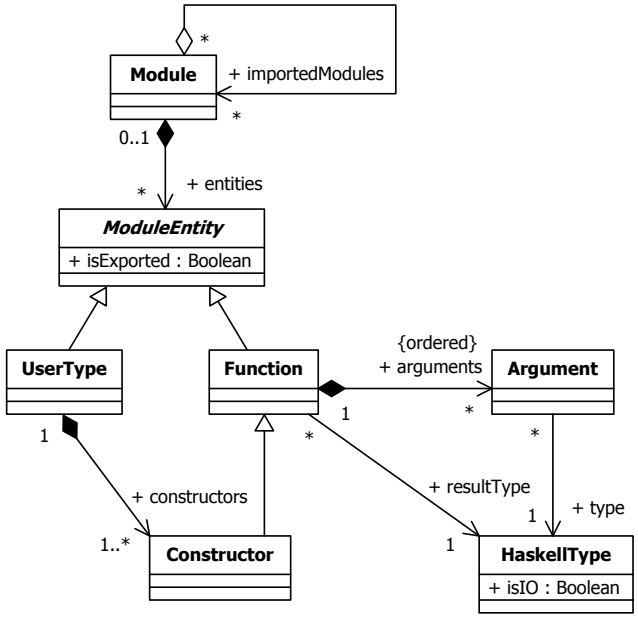

Fig. 2 Module contents

Types (Fig. 3) The most frequently used basic types in Haskell are: '`Bool`', '`Char`', '`String`', '`Int`', '`Integer`' (infinite-precision integers), '`Float`', '`Double`', and the *unit type* '`()`' (which is used when an IO action returns nothing). More complex types, like e.g. list types, are constructed from other types and are shown in Fig. 4. Polymorphic types are described in Haskell using type variables. For example, the type variable '`a`' in Fig. 1 represents any type. A user-defined type (an algebraic type) can be parameterized with one or more type variables and thus become a polymorphic one.[5]

[5] In fact, Haskell's type system is more sophisticated, but the simplified description presented here seems adequate to its purpose.

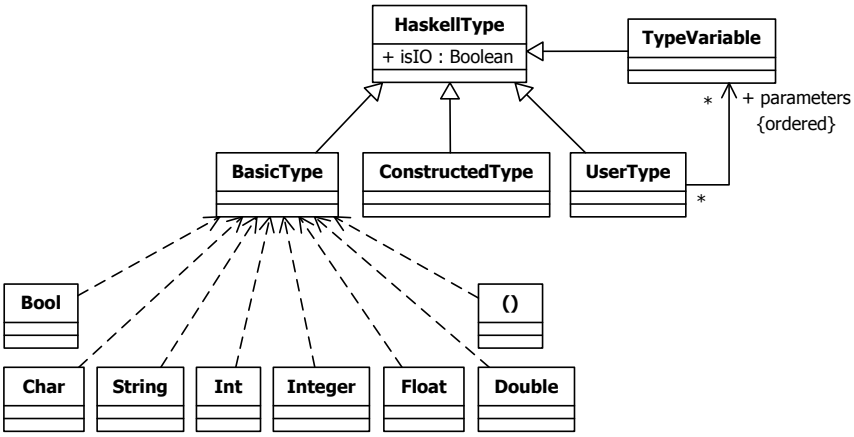

Fig. 3 Types

Constructed types (Fig. 4) Haskell offers also types which are constructed from other types (which themselves can be basic or constructed). These constructed types are:

- *list types* (e.g. '`[Char]`' is a list of characters),
- *tuple types* (e.g. '`(Int, Float)`' is an ordered pair, where the first element is an integer and the second is a real), and
- *function types* (e.g. '`Char -> Bool`' is a function which takes a character and returns a boolean result).

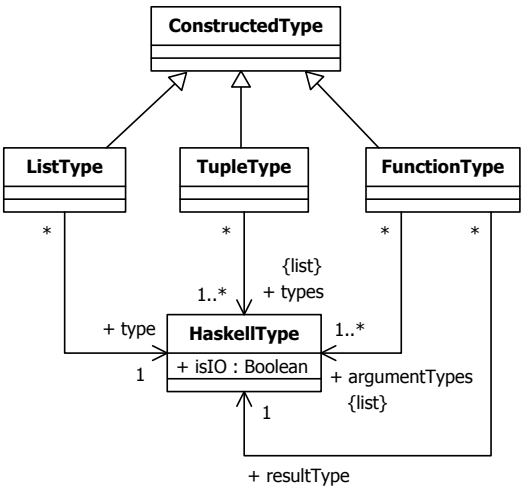

Fig. 4 Constructed types

Lists can hold an arbitrary number of elements, but these elements must all be of the same type. This contrasts with tuples, which hold only a fixed number of elements, but can be heterogeneous.

4 Profile

UML can be tailored to specific domains or programming environments by defining its dialect as a profile [14]. A UML profile identifies a subset of UML and defines *stereotypes* and constraints that can be applied to the selected UML subset. This section presents a UML profile for Haskell, but the profile presented can be also adopted to other functional languages. The profile consists of eight stereotypes which are a direct mapping of the concepts defined in the Haskell metamodel presented in the previous section. Table 1 depicts the relation between the stereotypes from the profile and the Haskell matamodel, as well as UML base elements for the stereotypes (i.e. elements to which the stereotypes can be applied).

Table 1 Mapping metamodel concepts to profile elements

Metamodel element	Stereotype	UML base element
Module	«Module»	Class
Function	«Function»	Operation
Function	«Value»	Attribute
Function	«IOAction»	Operation
UserType	«UserType»	Class or Parameterized class
Constructor	«Constructor»	Operation
Module.entities	«Contents»	Dependency
Module.importedModules	«Import»	Dependency

4.1 Stereotypes

In the following, the stereotypes in the profile and their use are briefly described.

Module This stereotype can be applied to a Class. Classes annotated with this stereotype represent Haskell modules. Functions defined in a given module can be then specified on an operation list of a Class or on an attribute list if a function is a value.

Function This stereotype is used on Operations to represent pure functions defined in a Haskell program. The default UML syntax for an operation is used:

```
name (parameter: parameter-type, ): return-type,
```

where ‘name’ is the name of the given function, ‘parameter’ is the name of the function argument, ‘parameter-type’ is the name of the type of that argument, and ‘return-type’ is the name of the type of the function result. Both the names of arguments, the names of their types and the name of the result type are optional (their appearance depends on how detailed the function is modeled). The name of the argument type and the name of the result type can be any Haskell type expressions. The only difference is for parameterized types, where the names of the type variables should be enclosed in angle brackets (< >), similar to the UML notation for *parameterized classes* (template classes) [14]. For example, ‘Stack a’ and ‘Either a b’ should be written as ‘Stack<a>’ and ‘Either<a,b>’, respectively. This is consistent to the way parameterized user-defined types are modeled (see the description for the *UserType* stereotype below).

Value This stereotype should be used to show a zero-argument pure function and it can be applied to an Attribute. The default UML syntax for an attribute is used:

```
name: type = value,
```

where ‘name’ is the name of the given value, ‘type’ is the name of the value type and ‘value’ is the given value. Only the name of the value is obligatory. The syntax rules for the name of the value type are the same like in the case of the *Function* stereotype.

IOAction This stereotype is used on Operations to represent IO actions. Its use is the same as of the *Function* stereotype. Zero-argument action should be also shown as an operation with *IOAction* stereotype. The operations annotated with the *Function* or *IOAction* stereotypes and the attributes with the *Value* stereotype can be declared only in a class having the *Module* stereotype.

UserType User-defined types should be shown as Classes annotated with a *UserType* stereotype. Constructors of such a type can be then specified on an operation list of a Class. For user-defined types that are parameterized with type variables the *UserType* stereotype should be applied to parameterized classes, where the number and the names of the parameters correspond to the number and the names of the type variables.

Constructor This stereotype can be applied to an Operation. Operations annotated with this stereotype represent constructors of a given user data type. The syntax for such an operation is the same as in the case of the *Function* stereotype. The only difference is that the arguments of the constructor do not have names. The operations annotated with the *Constructor* stereotype can be only declared in a class bearing the *UserType* stereotype.

Contents Functions defined in a module are specified on an operation list of a class representing this module. To show that a given user type is defined in a given module one should use a UML Dependency relationship with a *Contents* stereotype applied to it. This relationship should connect a class representing the module to a class representing the user type. The given user type may be contained in only one module.

Import This stereotype is to be applied to a Dependency relationship connecting two classes representing modules where one of these modules imports the other. This relationship should go from the class representing the importing module to the class representing the imported module. The information whether a module exports a function or data type defined in it should be shown as UML *visibility markers* [14] placed before the name of a function or data type. The marker '+' (public) denotes that the module element is exported and '–' (private) that it is not exported.

In Fig. 5 a model of a sample Haskell program from Fig. 1 is presented showing the application of some of the stereotypes.

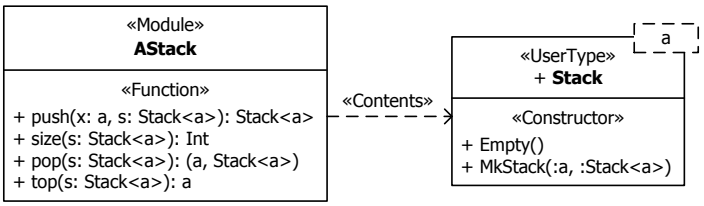

Fig. 5 A sample model

4.2 Constraints

A stereotyped UML element may have additional constraints beyond those of the base element [14]. Some of the additional constraints have been stated above, e.g. that the user type may be contained in only one module, and some other are omitted here. All such constraints come directly from the Haskell metamodel, what is an essential advantage of creating the metamodel of the language for which the UML profile is being defined.

5 Related Work

It seems that, so far, no work has been done on tailoring UML to model functional programs. In [17], rather than tailor UML to model Haskell programs, the translation from standard UML elements to Haskell is proposed. As the author himself admits, it results in some awkwardness in converting from the object-oriented to the functional paradigm and the Haskell code produced this way looks much more imperative than functional.

In general, not much work seems to have been done on modeling functional programs. In [15] a graphical modeling language is proposed, however, it is not related to UML in any way. Some *visual functional programming languages* [2, 12] have been also defined, but they focus on graphical representation of algorithms rather than abstract models of programs.

6 Conclusion and Further Work

The main idea behind this chapter is filling the gap in the area of graphical notations for modeling functional programs. From the practical point of view, it seems attractive to use a widely known UML notation with its extensive tool support, rather than define a new notation from scratch. For that reason, the work on defining a metamodel and a UML profile for the Haskell language has been undertaken. Some of the initial results of this work have been presented in this chapter. Further work will focus on broadening the scope of Haskell included in the profile, as well as on providing metamodel and profile implementations for popular UML modeling tools.

References

[1] Backus J (1978) Can Programming Be Liberated from the von Neumann Style?A Functional Style and Its Algebra of Programs. Communications of the ACM 21(8):613–641

[2] Cardelli L (1983) Two-dimensional syntax for functional languages. Proceedings of ECICS 82:139–151

[3] Harrop J (2008) F# for Scientists. Wiley-Interscience

[4] Hudak P (1989) Conception, Evolution, and Application of Functional Programming Languages. ACM Computing Surveys 21(3):359–411

[5] Hutton G (2007) Programming in Haskell. Cambridge University Press

[6] Jones SP (2003) Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press

[7] Warmer J, Kleppe A (1998) The Object Constraint Language: Precise Modeling With UML. Addison-Wesley

[8] Milner R, Tofte M, Harper R, MacQueen D (1997) The Definition of Standard ML (Revised). MIT Press

[9] Object Management Group (2003) MDA Guide Version 1.0.1 (omg/03-06-01)

[10] Object Management Group (2004) Metamodel and UML Profile for Java and EJB Specification (formal/04-02-02)
[11] Object Management Group (2010) UML 2.3 Superstructure Specification (formal/2010-05-05)
[12] Reekie HJ (1995) Realtime Signal Processing: Dataflow, Visual, and Functional Programming. PhD thesis, University of Technology at Sydney
[13] Remy D (2002) Using, Understanding, and Unraveling the OCaml Language. In: Barthe G (ed) Applied Semantics, Advanced Lectures. LNCS 2395:413–537, Springer-Verlag
[14] Rumbaugh J, Jacobson I, Booch G (2004) The Unified Modeling Language Reference Manual, Second Edition. Addison-Wesley
[15] Russell D (2001) FAD: A Functional Analysis and Design Methodology. PhD thesis, University of Kent at Canterbury
[16] Turner DA (1985) Miranda: A non-strict functional language with polymorphic types. In: Functional Programming Languages and Computer Architecture. LNCS 201:1–16, Springer-Verlag
[17] Wakeling D (2001) A Design Methodology for Functional Programs. In: Taha W (ed) Semantics, Applications, and Implementation of Program Generation. LNCS 2196:146–161, Springer-Verlag