PURe CAMILA

A System for Software Development using Formal Methods

February 25, 2005

Alexandra F. Mendes
alexandra@correio.ci.uminho.pt

João F. Ferreira
joao@correio.ci.uminho.pt

Abstract

This work consists in the re-implementation of CAMILA[1], which is a software development environment intended to promote the use of formal methods in industrial software environments. The new CAMILA, also called PURe CAMILA, is written in *Haskell* and makes extensive use of monadic programming. A small prototype interpreter was created, but the purpose of this work is to study the concepts behind such a tool.

Contents

1 Introduction **2**
1.1 Prototyping specifications . . . 3
1.2 Structure of this report . . . 4

2 System's architecture **5**
2.1 Why Haskell? . . . 6
2.2 Monads . . . 6
2.3 Monad transformers . . . 8

3 VDM **8**
3.1 Introduction . . . 8
3.2 Examples . . . 9

4 PURe CAMILA **10**
4.1 H++ . . . 11
4.1.1 Introduction . . . 11
4.2 Exceptions . . . 11
4.2.1 Introduction . . . 11
4.2.2 Implementation . . . 11

[1]http://camila.di.uminho.pt

4.2.3 Examples 12
4.3 DT Invariants 13
4.3.1 Introduction 13
4.3.2 Invariants in **VDM-SL** 13
4.3.3 Implementation 14
4.3.4 Examples 15
4.3.5 Alternatives 15
4.4 Partiality 16
4.4.1 Introduction 16
4.4.2 Partiality in **VDM-SL** 16
4.4.3 Implementation 17
4.4.4 Examples 18
4.4.5 Alternatives 18
4.5 State 19
4.5.1 Introduction 19
4.5.2 VDM++ 19
4.5.3 Implementation 20
4.5.4 Examples 21
4.6 Persistence 21
4.7 Creating object-oriented specifications 22
4.7.1 Examples 24

5 Examples **24**

6 Conclusion **27**
6.1 Limitations 27
6.2 Future work 28

A Haskell examples **30**
A.1 Stack Algebra 30
A.2 Stack Object 31

1 Introduction

Traditionally, when programmers (specially those who have no formal methods experience) want to verify some program's correctness they do tests. Debugging consists, basically, in correcting a program by executing it successively and correcting it when some test fails.

This approach is not effective and Edsger W. Dijkstra, an eminent computer scientist, has summarized the flaw in testing in the following quotation:

> Program testing can be used to show the presence of bugs, but never to show their absence.

An alternative to debugging is to use techniques which allow the verification of a program against its specification. Formal methods and formal specification give the

programmer a science of programming, where programs can be calculated from their specifications.

Generally, when using a formal approach, software development is divided into a **specification** phase, in which a mathematical model is built from the user requirements and a **implementation** phase in which such model is somehow converted to the final product. In a sense, software development, using formal methods, is much alike the "universal problem solving" strategy [Oli97]:

1. understand your problem;
2. build a mathematical model of it;
3. reason in such a model;
4. upgrade your model, if necessary;
5. calculate a final solution and implement it.

In the second step is where we start writing the initial specification. However, problems may arise because that specification will be based on informal user requirements and we can not prove its correctness. Since the formal methods of system specification and development have as mission the construction of reliable computer systems, several techniques have been developed to validate specifications, such as syntax and type checking. Another technique is to obtain an executable specification by transforming the original one [Muk97, Gau94], which is used by specification tools such as the IFAD VDM-SL Toolbox[ELL94] and CAMILA[ABNO97].

Originally, this technique was used mapping the specification into a functional or logic programming language, but dedicated tools such as the IFAD VDM-SL Toolbox have proved to be more effective [Muk97]. Still, several studies have been made in order to **translate** VDM-SL to functional programming languages [Muk97, BM93]. Although this is not a translation work, these translation studies are interesting, since they help comparing functional programming languages with specification systems in terms of syntax and semantic meanings.

1.1 Prototyping specifications

Prototyping a model based specification S means translating S into a program P which, though not necessarily satisfying efficiency or other non-functional constraints, is correct with respect to S. This allows the specification to be animated, which enables developers and end users to check early in the development stage that the specification is a valid representation of their requirements. Other advantages include[TM92]:

- the promise of a working prototype lessens the risk factor involved for a software purchaser: instead of waiting untill a full implementation is available, the prototype is a working model which gives a good indication of what the final product will be like;
- constructing a prototype helps in debugging the specification: even after proof obligations have been successfully discharged there may remain errors in the logic.

If the programming language in which a program P is created (call it PL) is well chosen, then prototyping P can be a semi-automatic process with a very high degree of certainty that P will be correct with respect to S. To be a good prototyping language, PL should contain constructs that are similar to the specification language (say SL) in which S is written. In effect, this means that the semantic gap between SL and PL should be small, i.e., data types, functions and operations should be easily translated from SL to PL.

The main deficiency of prototyped specifications is that they tend to be inefficient in time and space. This happens because, usually P is written following the same structure as S and the compilers of good candidates for PL (such as Haskell) tend to produce less efficient code than imperative languages.

Despite these drawbacks, *Haskell* seems a good candidate for PL. In section 2.1 we explain why we have used the *Haskell* language.

The main goal of this work is to study the design of a generic specification tool, and implement it using *Haskell*, a purely functional programming language. We propose a simple and modular architecture where each piece may be "disconnected" from the global system. **It's important to keep in mind that this is a preliminary approach and some aspects may change in the future.**

1.2 Structure of this report

The structure of the present document is as follows:

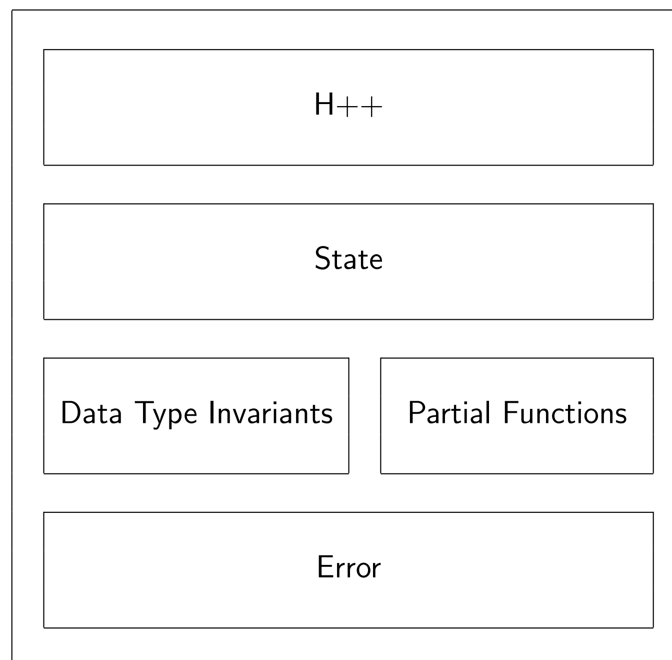
- **Section 2** describes the system's architecture and explains why we have used the functional programming language *Haskell*. It also provides an introduction to monadic programming methodology and monad transformers, including the description of their use in the components of the system.
- **Section 3** introduces the Vienna Development Method (VDM) and shows some **VDM-SL** and **VDM++** examples.
- **Section 4** explains the most important components of PURe CAMILA and their implementation;
  1. H++ (section 4.1): *Haskell* libraries implementing mathematical structures;
  2. Exceptions (section 4.2): the error mechanism;
  3. Data Type Invariants (section 4.3): constraints in the data types;
  4. Partiality (section 4.4): partial functions and pre-conditions;
  5. State (section 4.5): where we introduce the **VDM++**, a specification language that provides the mechanism to specify object oriented systems, and describe how we implement state;
  6. Persistence (section 4.6): explains a possible way of obtaining state persistence using the *hs-plugins* framework.

2 System's architecture

Concerning the system's architecture, we propose a **simple** and **modular** one where each piece may be "disconnected" from the global system.

Generally, we can describe the system architecture in the way it is presented in figure 1, where each block represents a system's feature.

Figure 1: PURe CAMILA architecture



The blocks illustrated in figure 1 are described as follows:

H++ - this block consists in a set of mathematical structures useful in formal specification. It will be described in section 4.1.

Invariants - this block refers to data type invariants. It will be described in section 4.3

Partiality - this block allows us to use partial functions in our specifications, using pre-conditions. It will be described in section 4.4

Error - this block represents the error mechanism which is necessary to every specification system. It will be described in section 4.2

State - this block allows us to build state models where we can define global variables and operations upon these variables. It will be described in section 4.5

It is important to allow the disconnection of these blocks from the global system, because there are cases where someone may not want them. For instance, it may be useful to turn off the data type invariants' verification or the functions' partiality (pre-conditions) to analyze how the system would behave in case of invalid input.

2.1 Why Haskell?

Haskell is a modern, standard, lazy, polymorphicly typed, purely-functional programming language. It has a module system and a type system which supports a systematic form of overloading and can express and enforce high assurance properties.

Writing and maintaining large software systems is difficult and expensive, but using a functional programming language like Haskell, it can be easier and cheaper.

Generally, functional languages are strongly typed, eliminating a lot of errors at compile time and at runtime. They are more intuitive, they offer more and easier ways to do the work and functional programs tend to be shorter, more stable, easier to maintain and easier to understand.

As we want to animate specifications, Haskell is a good option because it is superb for writing specifications which can be executed, due to its expressive syntax and a rich variety of built-in data types. Many other important Haskell's features are described in [Hug89, Syl], but the one we want to emphasize is the monadic programming approach which is described in the following section.

2.2 Monads

In this subsection, we will present the monadic programming methodology used in the implementation of CAMILA. Monads can be used for structuring functional programs and for implementing computational effects usually found in imperative programming languages, such as state variable updating[Oli01].

A very good introduction on monads is given in [Vis96], [Wad90] and [Wad92].

Definition 2.1. *A monad is an unary type M on which the functions*

$$unit : a \rightarrow Ma$$

and

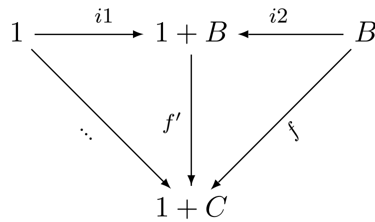
$$bind : Ma \rightarrow (a \rightarrow Mb) \rightarrow Mb$$

are defined in such a way that they obey the following three laws (the monad laws):

Left unit :	*(unit a) ‘bind‘ f = f a*
Right unit :	*m ‘bind‘ unit = m*
Associativity :	*m ‘bind‘ (λa. f a ‘bind‘ (λb. h b)*
	= (m ‘bind‘ (λa. f a)) ‘bind‘ (λb. h b)

In the context of this work, monads are used to implement three important mechanisms, described next:

Error - using the Exception Monad [Vis96, Wad90, Wad92]. Exception Monad presents a very interesting feature which is called *strict* composition: an exception produced by the *producer* function g is propagated to the output of the *consumer* function f. This feature is well explained in [Oli01] where the problem of composing partial functions is presented. Basically, if we want to compose the partial functions $1 + B \xleftarrow{g} A$ and $1 + C \xleftarrow{f} B$, we have to extend f to some f' capable of accept arguments from $1 + B$:



The most obvious instance of ... in the diagram above is $i1$ and this corresponds to *strict* composition. In practice, using this mechanism, exceptions may be propagated without programmer's intervention. Section 4.2 details the error mechanism implementation.

State - using the State Monad [Vis96, Wad90, Wad92]. Formal specifications are usually made of function and value definitions, together with a state and operations that may be performed on it. Implementing state in non-pure languages is natural, because they allow the definition of global variables. In pure functional languages, such as *Haskell*, computations which maintain state can be done using the State Monad. For a type S of states, the monad of state is defined by

$$\begin{array}{lll} type\ ST\ x & = & S \rightarrow (x, S) \\ unit\ x & = & \lambda s \rightarrow (x, s) \\ m\ \text{`}bind\text{`}\ f & = & \lambda s \rightarrow (f\ a)\ s' \mid (s', a) \leftarrow m\ s \end{array}$$

Values in the state monad are represented as transition functions from an initial state to a $(value, newState)$ pair and a new type definition is provided to describe this construct. Hence, the state monad might more appropriately be called a state transformer monad.

IO - using the Input/Output Monad [Vis96, Wad90, Wad92]. The system that will implement the specification mechanisms studied in this work will need to interact with the user and/or with the operating system. Hence, we need to use IO operations such as reading from standard input, writing to disk files and so on. Haskell provides IO functions using the monad IO.

Each of these three mechanisms by itself is indispensable but is not enough: we need to put them together in order to have their features at the same time. We have basically two ways of putting them together:

1. To create a new monad from scratch, and extend it with all the desired functionalities. This process is very laborious and time-consuming.
2. To use *monad transformers*. Monad transformers are type constructors which take monads as argument and yield new monads. We have used monad transformers which are introduced briefly in the following section.

2.3 Monad transformers

A monad transformer is a type constructor which constructs a new monad from another monad, allowing the creation of customized monads ([Vis96, KW93, LHJ95]). For each *base* monad described in the section 2.2, a corresponding monad transformer can be defined, which captures the same monad features. Combining it with an arbitrary monad enriches the monad with the features captured in the monad transformer. Thus, monad transformers can be used to combine in a single monad different monad features without demanding new data constructors or new monad functions.

Definition 2.2. *A monad transformer is a unary type constructor T which receives a monad as argument and yields a new monad and on which the function*

$$lift : Ma \rightarrow (TM)a$$

is defined in such a way that it obeys the following two laws (the monad transformer laws):

Unit lift : *lift* $(unit_m$ *a*$) = unit_{tm}$ *a*
Bind lift : *lift* (*m* '$bind_m$' $\lambda a.$ *f a*) = (*lift m*) '$bind_{tm}$' ($\lambda a.$ *lift* (*f a*))

So, if T is a monad transformer, then for all monad M, (TM) is also a monad. Monad transformers require the definition of the function $lift : M\ a \rightarrow (TM)\ a$ which encapsulates a monad M into a transformed monad TM. A monad transformer essentially "lifts" the operations of one monad into another.

3 VDM

The acronym VDM stands for Vienna Development Method and it's origin is related with the need to solve the problem of the systematic development of a compiler for the PL/1 programming language. This section highlights VDM's most important aspects for this work.

3.1 Introduction

The Vienna Development Method (VDM) was initially developed at the IBM Laboratory at Vienna in the early 1970's and it consists in a development method in which all design steps are expressed in a formal (mathematically based) notation [CO84].

VDM is one of the most mature methods and is used mainly for the formal specification and development of computing systems.

It consists of several elements[vdm]:

- a specification language called VDM-SL, which is used during specification and design phases of a computing system development;
- rules for data and operation refinement which allow one to establish links between abstract requirements specifications and detailed design specifications down to the level of code;

- a proof theory in which rigorous arguments can be conducted about the properties of specified systems and the correctness of design decisions.

The term "VDM" is sometimes used a little carelessly to mean the specification language only.

VDM is an example of a model based formal method where global variables (*state*) are associated with operations on these variables. The model is formal in the sense that it uses mathematically based notation (generally, the model uses simple data types like sets, lists or mappings) and abstract, in the sense that it is free from details concerning the eventual implementation (eg. efficiency and memory usage)[CO84].

For example, a specification of a bank system would contain a mapping from account number to account information (holders, ammount, etc.) along with operations to open new accounts, credit money, etc. Basic types like natural numbers, characters, and type constructors like sets and maps, are provided "for free" in VDM-SL.

Generally, a VDM development is made up of state descriptions at sucessive levels of abstraction and of implementation steps which link the state descriptions[LS90].

Several studies show that VDM is a good choice to help programmers in the early phases of the software development cycle, since it allows to analyze safety conditions and properties [ALR98, LS90]. Another good argument to use VDM is that the VDM Specification Language (VDM-SL) achieved ISO Standardisation in 1996.

Obviously, VDM wouldn't be widely used if there weren't tools supporting it. For this work we have only used and studied *IFAD VDM Tools* [ELL94].

For more informations about VDM we recommend the consult of [Lar94, Jon90].

3.2 Examples

In this section we will show some VDM-SL and VDM++ examples. These examples were animated using the *IFAD VDM Tools*.

VDM-SL: Stack of odd integers

```
types
Stack = seq of int
inv s == forall i in set (elems s) & i mod 2 <> 0

functions
empty: Stack -> bool
empty(s) == s = [];

push: int * Stack -> Stack
push (n,s) == [n]^s;

pop: Stack -> Stack
pop(s) == tl s
pre not empty(s);
```

```
top: Stack -> int
top(s) == hd s
pre not empty(s);

add: Stack -> Stack
add(s) == [s(1) + s(2)]^(tl (tl s))
pre len(s) >= 2;

values
stack1 = [1,3,5];
stack2 = []
```

VDM++: State

```
class stackObj is subclass of stackAlg

instance variables

   public stack: Stack := init();

operations

   public CLEAR: () ==> ()
   CLEAR() == stack := init();

   public PUSH: A ==> ()
   PUSH(a) == stack := push(a,stack);

   public POP: () ==> A
   POP() == def r = top(stack)
            in (stack := pop(stack);
                return r)
   pre not empty(stack);

   public TOP: () ==> A
   TOP() == return top(stack)
   pre not empty(stack);

end stackObj
```

---

4 PURe CAMILA

In this section we will describe how we have implemented the most important components of CAMILA specification system.

4.1 H++

In this section we'll present H++, the Haskell libraries which implement mathematical structures used in formal specifications, such as sets, mappings, relations, etc.

4.1.1 Introduction

In the "universal problem solving" strategy presented in the introduction (page 2), one of the steps was to build a mathematical model of the problem in order to reason in such a model. This mathematical model can only be built if we use mathematical structures, such as sets, mappings, lists, etc.

Specification languages must permit abstraction from details while preserving essential properties. Mathematics are used to define more precisely the abstract model.

The advantage of this mathematics is that it provides tools for formal reasoning about specifications, which can then be examined for completeness and consistency. [Vie93]

These libraries are detailed in [MFP04]

4.2 Exceptions

In this section we will present a way to represent exceptions.

4.2.1 Introduction

Exception handling is a vital piece in the system. We will want the system to throw an exception every time a data type invariant or a precondition is violated.

In the next section we will detail how we've implemented the error mechanism.

4.2.2 Implementation

As already mentioned in section 2.2, we have used error monad to implement the error mechanism. *Haskell* provides libraries that already define error monad. Using these libraries, we just have to define the error type, declare it as instance of Error class and finally, to create the type that will hold values or exceptions. From this point, all the functions that can throw errors must have as codomain the last type, capable of holding values or errors.

More formally, supposing $CamilaError'$ is the type that defines the error, a total function $B \xleftarrow{f} A$ will be extended to $CamilaError \xleftarrow{f} A$, where $CamilaError = CamilaError' + B$, meaning that it can generate an exception **or** yield a value of type B.

Defining $CamilaError'$ depends on the needs and on the debug information we want to hold. For now, our type is defined in the following way

$$data\ CamilaError' \ = \ Err\ \{\ reason :: String\}$$

holding just the error message. In the future, we may want to hold detailed information used for debugging.

What about $CamilaError$ type? It's obvious that $CamilaError$ is a coproduct ($CamilaError'+B$), implemented in *Haskell* as $Either$. Hence, we define $CamilaError$ as follows

$$type\ CamilaError\ a\ =\ Either\ CamilaError'\ a$$

Using this type, we will represent an error as $Left\ (Err\ "Error\ Message")$ and a value $a \in A$ as $Right\ a$.

4.2.3 Examples

In this section we can see some examples, using *ghci*, that demonstrate the error propagation mentioned in section 2.2.

Consider the following functions:

```
import Camila.Error

f1 :: Int -> CamilaError Int
f1 x | x < 0     = fail $ show x ++ " is a negative number"
     | otherwise = Right (x+1)

comp_ok :: CamilaError Int
comp_ok = do x <- f1 1
             y <- f1 2
             return (x+y)

propagate :: CamilaError Int
propagate = do x <- f1 1
               y <- f1 (-1)
               z <- f1 (-2)
               return (x+y)
```

Function $f1$ throws an error if its argument is a negative number and behaves as the successor function if its argument is a natural number. Given two monadic values $m1$ and $m2$, it's possible to "sequence" them, obtaining another such value. In *Haskell*, we usually use the *do-notation* to sequence monadic values. The function $comp_ok$ is an example of this sequencing mechanism: it holds in x the value of $f1(1) = 2$, in y the value of $f1(2) = 3$ and it returns value $2 + 3 = 5$. Since $return$ corresponds to function $unit$ defined in section 2.2, $comp_ok$ should return $Right\ 5$. In fact:

```
*Main> comp_ok
Right 5
```

What about error propagation? Function $propagate$ is a clear example: when $f1$ evaluates value -1, it will throw an exception and $propagate$ will never evaluate the next "steps".

```
*Main> propagate
Left Error: -1 is a negative number
```

4.3 DT Invariants

Data type invariants are a very important part of any formal specification system. In the next sections we'll discuss why they are important and how we've implemented them.

4.3.1 Introduction

If a data type contains values which should not be allowed, then it's desirable to restrict its values by means of an invariant. The result of such a constraint is that the type is restricted to a subset of its original values. The elements of this subset are the ones that respect the predicate that defines the invariant, ie, the ones who make the predicate expression true.

In the scope of this work, a data type A constrained by an invariant ϕ (different from the always true predicate – $\lambda b.True$) is represented by A_ϕ. If $\phi = \lambda b.True$ then $A_\phi = A$ and we ommit ϕ.

A constrained data type A_ϕ is therefore handled defining a predicate ϕ that yields true for all acceptable values of type A. For values $a \in A$ with $\neg\phi(a)$, the data type invariant is violated.

When specifying a system, one may want to establish relationships between its components that have to be fixed throughout its execution lifetime. Data type invariants provide a way of defining properties on types.

In sum, data type invariants are a very important mechanism that help us think about properties of a program. Thinking about datatype invariants will help to improve the data type implementation.

4.3.2 Invariants in VDM-SL

In **VDM-SL** we would define a data type invariant as follows:

```
Stack = seq of int
inv s == forall i in set (elems s) & i mod 2 <> 0
```

In this example, we are defining a `Stack` as a sequence of integers, but nor all sequences of integers are contained in type `Stack`. In fact, the invariant expression validates only sequences of odd integers.

4.3.3 Implementation

Implementing data type invariants in *Haskell* may be done in several different ways. The main idea is that, for a given data type, we must define a predicate which defines its invariant and a way to check if, for certain values of that type, the invariant is true.

Basically, we have seen two alternatives for invariants implementation:

1. Definition of a function for each data type which defines and check the predicate. This is the **IFAD VDM-SL Toolbox** approach: they define a inv_T function for type T. This approach was not the chosen one and therefore it will be described in section 4.3.5.
2. Definition of an overloaded function for each data type. Using *Haskell*'s type classes system, invariants' definition become very elegant. This is the chosen approach to implement data type invariants and is described in this section.

So, we have defined a class named *CData*[Nec05] (**C**onstrained **Data** Type) which defines two functions:

- *inv :: a → Bool*: this is the predicate that we need to define. The default value is true.
- *inv' :: a → CamilaError a*: this is the function that will test the predicate and, in the case of fail, returns an error.

So, in order to define a constrained data type, we just have to declare that type as instance of class *CData* and then define the predicate *inv*. Next, we present how we can write in *Haskell* the example written above in **VDM-SL**:

```
-- (1) Datatype definition
type Stack = [Int]

-- (1.1) Datatype invariant
instance CData Stack where
   inv s = filter odd s == s
```

As we can see, the data type definition is straightforward. The invariant definition consists in declaring an instance of *CData* class for the type *Stack* and defining the function *inv*.

The *CData* class is defined as follows:

```
class CData a where
    inv :: a -> Bool
    inv a = True
    inv' :: a -> CamilaError a
    inv' a = if (inv a) then return a else fail "Invariant violation"
```

If we want to check the invariant of a given data type, we just have to run *inv'* function. As an example, we can define *empty* as follows:

```
empty :: Stack -> CamilaError Bool
empty s = do inv' s
             return (s == [])
```

4.3.4 Examples

In this section we can see some execution examples, using *ghci*. The haskell source is in appendix (page 30).

```
camila@host:~/libs$ make ghci top=Camila/Examples/StackAlg.hs
...
...
Prelude Camila.Examples.StackAlg> empty [1]
Right False

Prelude Camila.Examples.StackAlg> empty [1,2]
Left Error: Invariant violation

Prelude Camila.Examples.StackAlg> pop [1,3]
Right [3]

Prelude Camila.Examples.StackAlg> pop [1,2]
Left Error: Invariant violation
```

4.3.5 Alternatives

As we've seen in section 4.3.3, we can define a function for each data type which defines and checks the predicate.

The **IFAD VDM-SL Toolbox** approach is to define a function $inv_T :: t \rightarrow Bool$ where $t \in T$, which for a given allowed value t returns true.

To compare these two alternatives, lets define a function f that will check a certain invariant ϕ.

We call this function f_inv. Using this last alternative, we could define it in the following way:

$$f_inv\ x \quad = \quad if\ \phi(x)\ \ then\ f\ x \\ else\ (error\ "Data\ type\ invariant\ violation") \tag{1}$$

Using the first alternative, we would define f_inv as follows:

$$f_inv\ x \quad = \quad do \quad inv' x \\ \qquad\qquad\qquad\qquad f x \tag{2}$$

We can not refute the fact that this second approach is much more elegant than the first one.

4.4 Partiality

In this section we will show how we implemented partial functions (and pre-conditions).

4.4.1 Introduction

A partial function is a function which is not defined for some of its domain. For instance, division is (usually) a partial function since anything divided by 0 is undefined.

Partiality of a function f is usually handled defining a predicate (we call it pre-condition) pre that yields true for all values on which f is defined. For values x with $\neg pre(x)$, we say that the pre-condition is not valid and f can not produce any result.

So, pre-conditions can be defined as constraints on the input for which an operation is defined. They define the input values for which valid outputs can be expected, ie, they ensure normal behaviour of the function.

Having no pre-condition is equivalent to a pre-condition of true. By definition, when a pre-condition fails it means that the operation is prevented from executing.

A very interesting point of view is that a pre-condition $\mathbb{P}$ can be seen as a **local data type invariant** for the function's domain. For instance, in the following definition,

$$f : A \rightarrow B \tag{3}$$

$$f(a) = b \tag{4}$$

$$pre\ \mathbb{P}(a) = ... \tag{5}$$

$\mathbb{P}$ can be seen as type A's invariant **only** in the scope of f's definition (4).

4.4.2 Partiality in VDM-SL

In **VDM-SL** we would define a pre-condition as follows:

```
top: Stack -> int
top(s) == hd s
pre not empty(s);

add: Stack -> Stack
add(s) == [s(1) + s(2)]^(tl (tl s))
pre len(s) >= 2;
```

In this example, function *top* will yield a result if the input stack is not empty. If the stack is empty, then an error will occur saying that the pre-condition was violated.

The function *add* has also a pre-condition associated which restricts its input to stacks with more than one element.

4.4.3 Implementation

In the specification presented in page 9 we can see three partial functions: *pop*, *top* and *add*. Their partiality is defined using the expression *pre*. For instance, *pop* defines *pre* as *pre not empty(s)*, i.e., the stack received by *pop* can't be empty.

We show two alternatives for defining these pre-conditions:

1. Definition of a mechanism similar to the one presented in section 4.3, exploring the idea presented above, in which a pre-condition can be seen as a **local** data type invariant.
2. Definition of a function for each function which defines the predicate. This is the **IFAD VDM-SL Toolbox** approach: they define a pre_F function for function F. This alternative was not the chosen one and it will be presented in section 4.4.5.

So, in section 4.3 we have seen a way of defining data type invariants. We can use the same idea for partial functions, since a pre-condition is just a constraint on the function's argument.

Following this idea, we have defined a class named *Partial* defined as follows:

```
class Partial a where
    pre :: a -> Bool
    pre a = True
    pre' :: a -> CamilaError a
    pre' a = if (pre a) then return a else fail "Pre-condition violated!"
```

This class is very similar to *CData* class. Actually, the only differences are the name of the functions and the error message!

A problem arises when we try to define pre-conditions for partial functions using this method: we can't create more than one instance of Partial for the same type. So, if we have more than one partial function with the same type argument, we have to define more than one instance of Partial for that type. Since this is not possible, we decided to create a *local* data type, similar to the original one, but with a different name.

Briefly, to create a pre-condition:

1. We create a *local* data type, similar to the type argument, which name is *Datatype_function* (eg. *Stack_pop*, *Stack_top*, *Stack_add*);
2. We define an instance of *Partial* to this *local* data type.

For example, we would define *pop* as follows:

```
-- First, we create a "local" data type
data Stack_pop = Stack_pop Stack deriving Show

-- Second, we define it as instance of Partial
instance Partial Stack_pop where
   pre (Stack_pop s) = not $ s == []

-- Pop is defined as follows:
pop :: Stack -> CamilaError Stack
pop s = do pre' (Stack_pop s)
           s1 <- Right (tail s)
           return s1
```

4.4.4 Examples

In this section we can see some execution examples, using *ghci*. The haskell source is in appendix (page 30).

```
Prelude Camila.Examples.StackAlg> pop [1,3]
Right [3]

Prelude Camila.Examples.StackAlg> pop []
Left Error: Pre-condition violated!

Prelude Camila.Examples.StackAlg> add [1,3,5]
Right [4,5]

Prelude Camila.Examples.StackAlg> add [1]
Left Error: Pre-condition violated!
```

4.4.5 Alternatives

As said before, another way of defining partial functions is to translate the function without the pre-condition directly and then create a second, total, function that will check the condition. The latter, in case of success, will call the partial one, otherwise it returns an error.

Using this method, *pop* would become:

```
-- Translated directly
pop' :: Stack -> Stack
pop' s = tail s

-- We define a total function that does pre-checking:
pop_total :: Stack -> CamilaError Stack
pop_total s = if (pre s) then return (pop' s)
```

```
                    else fail "Pre-condition violation!"
where pre = (\x -> not $ x == [])
```

Comparing the two alternatives as we did before on page 15, we define the function that will check a pre-condition ϕ for f in the following way:

$$\begin{array}{rcll} f_pre\ x & = & if\ \phi(x) & then\ f\ x \\ & & & else\ (error\ "Pre-condition\ violated") \end{array} \tag{6}$$

Using the first alternative, we would define f_pre as follows:

$$\begin{array}{rcll} f_pre\ x & = & do & pre'x \\ & & & fx \end{array} \tag{7}$$

4.5 State

In this section we will present a way to represent VDM-SL state variables using monads.

4.5.1 Introduction

VDM-SL specifications are usually made of function and value definitions, together with a state and operations that may be performed on it. Implementing state in non-pure languages is natural, because they allow the definition of global variables. Haskell doesn't allow these type of variables, but we may use monads to represent a specification' state.

Up to now we have considered only functions which, given one or more input values, would produce some result. This result could then be the input for other functions (function composition). Although this form of specification is theoretically sufficient, it is unrealistic for most computing systems.

One of the most important steps in a model based specification is to capture the central object or objects of the problem and map those objects to data structures. In Information Systems applications, for example, the central object might be a library system or some other large data base[TM92].

This central object is called the system state, and instances of it are called states. Often an operation may only need to access or change a small part of the state; all other parts of the state are implicitly assumed to be unchanged.

4.5.2 VDM++

IFAD VDM-SL Toolbox have an interpreter for VDM++, which is a formal specification language intended to specify object oriented systems. The language is based on VDM-SL, and has been extended with class and object concepts, which are also present in languages like Smalltalk-80 and Java. This combination facilitates the development of object oriented formal specifications [IFA00].

The specification presented in 3.2 can be animated with that interpreter, like in the following example:

```
> create s := new stackObj()
> create r := new stackObj()
> print s.PUSH(1)
> print r.PUSH(3)
```

In this case, two instances have been created (r and s) and operation $PUSH$ was executed in each one.

4.5.3 Implementation

We have used state monad to implement state variables and state reading and updating. *Haskell* provides libraries that already define the state monad and functions that will help state manipulation. In fact, the use of state monad is "disguised": we have used state monad transformer applied to monad IO. This new monad (state + IO) was then applied to the error monad transformer yielding another monad, with the features we need (error + state + IO).

Using state and error monad transformers, already defined in *Haskell* libraries, the IO monad and the error structure ($CamilaError'$), we have defined the following data type,

$$\begin{aligned}
\mathit{type\ InterpState\ a} &= \mathit{ErrorT\ CamilaError'\ (StateT\ CamilaState\ IO)\ a} \\
&= \mathit{ErrorT\ (StateT\ CamilaState\ IO\ (Either\ CamilaError'\ a))} \\
&= \mathit{ErrorT\ (StateT\ (CamilaState \rightarrow IO\ (Either\ CamilaError'\ a, CamilaState)))}
\end{aligned}$$

where $CamilaState$ is:

$$\begin{aligned}
\mathit{data\ CamilaState} = \mathit{CamilaState}\ \{\ & \mathit{checkDTInv} :: \mathit{Bool}, \\
& \mathit{checkPre} :: \mathit{Bool}, \\
& \mathit{instances} :: \mathit{FiniteMap\ String\ (String, Dynamic)} \\
\}\ &
\end{aligned}$$

$CamilaState$ is the data type that holds state information. For the moment, our state has three values:

- $checkDTInv$ - if true, system will check data type invariants;
- $checkPre$ - if true, system will check pre-conditions;
- $instances$ - map that holds all instances: the keys are instances' identifiers, the values are those instances' state.

More precisely, the values of map $instances$ are a pair $(String, Dynamic)$ where the first component represents the "class" name (used by the interpreter) and the second component represents the instance's state. We have wrapped the state with data type Dynamic, in order to have an heterogeneous finite map. We will have to use injection $toDyn$ and projection $fromDyn$ to get the "real" state.

A function operating on the state will be of type $InterpState\ a$, where a is the value it returns. Since we have used *Haskell* libraries, we have access to some functions that help changing variables and getting their state

available some functions that help us to change and get state variables. For instance, the function that creates a new instance (and returns the new instances space) can be defined as follows:

```
create :: (Typeable a) => String -> String -> a ->
                          InterpState (FiniteMap String (String,Dynamic))
create id t obj = do modify (\s -> s { instances = addToFM (instances s) id (t,(toDyn obj)) })
                     insts <- gets instances
                     return insts
```

4.5.4 Examples

In this section we can see some execution examples, using *ghci*. The haskell source is in appendix (page 31).

```
camila@host:~/libs$ make ghci top=Camila/Examples/StackObj.hs
...
...

Prelude Camila.Examples.StackObj> exec obj_prog1
Top: 5
Stack: [5,3,1]
()

Prelude Camila.Examples.StackObj> exec obj_prog2
ERROR: Invariant violation

Prelude Camila.Examples.StackObj> exec obj_prog3
ERROR: Pre-condition violated!
```

4.6 Persistence

How that we have state, we want to keep it from computation to computation, i.e., we want state persistence. In order to have persistence and to animate specifications defined using the methods described in this report, we have built a small interpreter

which connects all blocks. This interpreter is based in *hs-plugins*[2], a framework for loading plugins written in Haskell into an application in runtime[PSSC04].

At this moment, the interpreter has as only purpose the animation of the concepts explored throughout this report. The interpreter allows us to load our specifications, to create new instances and to run operations on these instances. It also allows the program to have persistence, i.e., to hold the state from computation to computation and while the user interacts with it. We'll show some examples in the section 4.7.1.

There is a important issue about the instances structure presented above, which is defined as

$$instances :: FiniteMap\ String\ (String, Dynamic)$$

The problem is that, although *Haskell* libraries provide the data type Dynamic, we could not use it in the interpreter. We had to use the Dynamic provided with *hs-plugins* package. After several IRC and email discussions with Donald Bruce Stewart (hs-plugins author), we found out that Data.Dynamic fails due to a static/dynamic typing key problem. The solution is to use AltData.Dynamic provided with *hs-plugins*. The only difference is that new types have to have an instance Typeable, rather than just "deriving Typeable".

This small interpreter is very interesting in the way it makes possible to see the connection between the mechanisms described in this report. User interaction is facilitated because we have the possibility of running *Haskell* functions received as $Strings$. This way, we can create customized commands "on-the-fly" and run them.

We consider it very slow, but, we don't think it's for the monad use. In fact, we think the reasons are:

- Every time we evaluate an expression, *hs-plugins* framework creates a temporary file with some extra code, runs it and shows the result. We consider that the I/O operations associated contribute significantly to time degradation;
- Every time we need to operate in an arbitrary instance state, we have to:
  - get dynamic representation of the state;
  - get the "real" state using projection $fromDyn$
  - run the desired operation in the above state: this is accomplished creating a $String$ with the command we would execute (using $runInstance$ function with the above state), running it with *hs-plugins* functions and updating the $instances$ state with the new state.

4.7 Creating object-oriented specifications

When creating an object-oriented specification, some systematic steps must be done.

Let's build such a specification step-by-step. The specification we'll build is the one of a Stack Folder. Basically, we have a left stack and a right stack that, together, represent a book: the read pages are in the left stack and the unread pages are in the right stack. In section 5 we will present the same example.

[2] `http://www.cse.unsw.edu.au/~dons/hs-plugins`

First, we need to define the module name and to import the modules we need.

```
module Camila.Examples.Folder where

import Camila.Prelude
import Camila.Examples.StackAlg
import AltData.Typeable
```

The last import (AltData.Typeable) must be present in every specification, because we need to define the instance variables as instance of class AltData.Typeable.

Next, we must define the instance variables and the initial state:

```
-- Instance Variables
data FolderIVars = FolderIVars { stateFolder :: (Stack,Stack) } deriving Show

-- Initial State
initialFolder = FolderIVars { stateFolder  = ([],filter odd [1..4]) }
```

A important point is that specifications will have to have certain elements with fixed names. Supposing the module name is $Module$, we will have to have the following fixed names:

Instance variables - must begin with the module name, followed by the $IVars$. Examples: ModuleIVars, FolderIVars, StackIVars.

Initial state - must begin with $initial$, followed by the module name. Examples: initialModule, initialFolder, initialStack.

State variables - must begin with $state$, followed by the module name and it must be a tuple with all the desired values. Examples: stateModule, stateFolder, stateStack.

We also have to define the instance variables type as instance of class Typeable, so that the interpreter can use injection $toDyn$ on that type. It's not very difficult, as we can see:

```
-- We have to define FolderIVars as instance of AltData.Typeable to
-- allow the interpreter to make dynamic values
instance Typeable FolderIVars where
    typeOf _ = mkAppTy (mkTyCon "FolderIVars") []
```

Finally, we must define the operations. Every operation op must be of type

$$op :: ST\ ivars\ a$$

where $ivars$ is the type of the instance's variables and a is the operation's return type. For instance, the operation that returns the current page of a book is:

```
oppage :: ST FolderIVars Int
oppage = do (lpages,rpages) <- gets stateFolder
            top <- liftE $ top rpages
            return top
```

4.7.1 Examples

Examples of the interpreter working can be seen in section 5

5 Examples

In this section we will show a specification based on the mechanisms described in this report.

The example we present in this section is the one of a Stack Folder. Basically, we have a left stack and a right stack that, together, represent a book: the read pages are in the left stack and the unread pages are in the right stack. In this book we can:

- read one page (**opread**): we pop one element of the right stack and push it to the left stack;
- get one page back (**opback**): we pop one element of the left stack and push it to the right stack;
- get the current page (**oppage**): we just have to return the top element of the right stack.

In *Haskell*, we could define this example in the following way:

```
{-# OPTIONS -fglasgow-exts #-}
module Camila.Examples.Folder where

import Camila.Prelude
import Camila.Examples.StackAlg
import AltData.Typeable

-- Instance Variables
data FolderIVars = FolderIVars { stateFolder :: (Stack,Stack) } deriving Show

-- Initial State
initialFolder = FolderIVars { stateFolder  = ([],filter odd [1..4]) }

-- We have to define FolderIVars as instance of AltData.Typeable to
-- allow the interpreter to make dynamic values
instance Typeable FolderIVars where
    typeOf _ = mkAppTy (mkTyCon "FolderIVars") []

-- Operations
```

```
-- opread: reads one page

-- opread has a pre-condition: when we reach the end of the book, we
-- can not read another page
data Folder_opread = Folder_opread FolderIVars deriving Show

instance Partial Folder_opread where
    pre (Folder_opread f) = snd (stateFolder f) /= []

opread :: ST FolderIVars ()
opread = do (lpages,rpages) <- gets stateFolder
            folder <- get
            liftIO $ putStrLn $ "Before: " ++ show folder -- just to check state
            liftE $ pre' (Folder_opread folder) -- check pre-condition
            rpages2 <- liftE $ pop rpages
            lpages2 <- liftE $ push (head rpages) lpages
            modify(\s -> s { stateFolder = (lpages2,rpages2) })
            folder <- get
            liftIO $ putStrLn $ "After: " ++ show folder -- just to check state

-- opback: gets back one page

-- opback has a pre-condition: when we reach the beginning of the
-- book, we can not go back any further
data Folder_opback = Folder_opback FolderIVars deriving Show

instance Partial Folder_opback where
    pre (Folder_opback f) = fst (stateFolder f) /= []


opback :: ST FolderIVars ()
opback = do (lpages,rpages) <- gets stateFolder
            folder <- get
            liftIO $ putStrLn $ "Before: " ++ show folder -- just to check state
            liftE $ pre' (Folder_opback folder) -- check pre-condition
            lpages2 <- liftE $ pop lpages
            rpages2 <- liftE $ push (head lpages) rpages
            modify(\s -> s { stateFolder = (lpages2,rpages2) })
            folder <- get
            liftIO $ putStrLn $ "After: " ++ show folder -- just to check state


-- oppage: returns the current page

oppage :: ST FolderIVars Int
oppage = do (lpages,rpages) <- gets stateFolder
            top <- liftE $ top rpages
            return top
```

Next, we'll see the interpreter in action. First, let's start it:

```
camila@host(~)$ ./icamila
  ____                _ _
 / ___|__ _ _ __ ___ (_) | __ _
| |   / _` | '_ ` _ \| | |/ _` |     Camila - Formal Specification System
| |__| (_| | | | | | | | | (_| |     http://camila.di.uminho.pt
 \____\__,_|_| |_| |_|_|_|\__,_|     Type :? for help

Loading package base
Loading package lang
 ... linking ... done
camila>
```

Next, we need to load the Folder example:

```
camila> :l Camila.Examples.Folder
camila>
```

Now, we have to create a Folder instance:

```
camila> create f := new Folder
camila> print f
<FolderIVars>
camila>
```

As we can see, a new folder instance has been created. Using command *print* we can see the type of the instance.

Now, we can run the defined operations on the instance f:

```
amila> eval f opread
Before: FolderIVars {stateFolder = ([],[1,3])}
After: FolderIVars {stateFolder = ([1],[3])}
()
camila> eval f opread
Before: FolderIVars {stateFolder = ([1],[3])}
After: FolderIVars {stateFolder = ([3,1],[])}
()
camila>
```

As we can see, we are able to run operations on instances and the interpreter allows persistence! In fact, if we run *opread* once again, we should get a pre-condition error.

```
camila> eval f opread
Before: FolderIVars {stateFolder = ([3,1],[])}
Pre-condition violated!
camila>
```

As expected, the pre-condition was violated. It's very interesting to note that after the error ocurrence, the rest of the operations are not evaluated. This is due to the already mentioned monadic error propagation.

6 Conclusion

Before starting the project, the general idea was that stateless formal specifications would be easily "converted" to *Haskell* and object oriented specifications would be a little more complicated. It's proved that this initial idea was right: *Haskell* is so expressive that the stateless specifications are almost directly translated. The object oriented specifications need a little more work to be translated.

It's important to note that the framework created allows, in a systematic and elegant way, the animation of formal specifications. The systematic property is important in the way it facilitates the conversion of VDM-SL to our framework. We believe this conversion is not difficult and won't offer great resistance.

The use of *Haskell* type classes and the monadic approach make the methodology presented more elegant and structured. In pages 15 and 19 we've presented the difference between using error monad and the traditional "if ... then ... else" code, and we believe that the monadic alternative is better, not only because it's shorter but also because it's more structured. Using this alternative, to check or not a data type invariant or a pre-condition is just determined by the existence of inv' or pre'.

Finally, the interpreter we've created is significant in the sense it proves that it's possible to integrate all the concepts presented.

6.1 Limitations

Although we have demonstrated *Haskell*'s capability to create programs that animate formal specifications, there are several limitations in this work:

Debugging information - we consider debugging information important, and at this moment, the only debugging information available is the error type;

Partiality code - to create a pre-condition, we need to create a local data type and then define it as instance of class Partial. This methodology is not very pleasant and if the *Haskell* prototype is not automatically generated, it can be a very tedious job;

Special cases - we don't have much experience in formal specification and it's possible that some cases may be difficult to prototype, using the presented methodologies.

6.2 Future work

As future work, we think it would be important to:

- create VDM-SL frontends capable of converting VDM-SL specifications to *Haskell* code following the methodologies presented;
- create a good interpreter with automatic module loading and compiling to animate the specifications;
- study the possibility of defining every mechanism as a monad transformer (see [LHJ95]).

References

[ABNO97] J. J. Almeida, L. S. Barbosa, F. L. Neves, and J. N. Oliveira. Camila: Prototyping and refinement of constructive specifications. In *AMAST '97: Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology*, pages 554–559. Springer-Verlag, 1997.

[ALR98] Sten Agerholm, Pierre-Jean Lecoeur, and Etienne Reichert. Formal specification and validation at work: a case study using vdm-sl. In *FMSP '98: Proceedings of the second workshop on Formal methods in software practice*, pages 78–84. ACM Press, 1998.

[BM93] Paulo Borba and Silvio Meira. From vdm specifications to functional prototypes. *J. Syst. Softw.*, 21(3):267–278, 1993.

[CO84] Geert B. Clemmensen and Ole N. Oest. Formal specification and development of an ada compiler - a vdm case study. In *ICSE '84: Proceedings of the 7th international conference on Software engineering*, pages 430–440. IEEE Press, 1984.

[ELL94] René Elmstrom, Peter Gorm Larsen, and Poul Bogh Lassen. The ifad vdm-sl toolbox: a practical approach to formal specifications. *SIGPLAN Not.*, 29(9):77–80, 1994.

[Gau94] Marie-Claude Gaudel. Formal specification techniques (extended abstract). In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 223–227. IEEE Computer Society Press, 1994.

[Hug89] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[IFA00] IFAD. The ifad vdm++ language. Technical report, 2000. User Manual.

[Jon90] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990.

[KW93] David J. King and Philip Wadler. Combining monads. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 134–143. Springer-Verlag, 1993.

[Lar94] Peter Gorm Larsen. The VDM bibliography. Technical report, Forskerparken 10, DK-5230 Odense M, Denmark, January 1994.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343. ACM Press, 1995.

[LS90] Yves Ledru and Pierre-Yves Schobbens. Applying vdm to large developments. In *Conference proceedings on Formal methods in software development*, pages 55–58. ACM Press, 1990.

[MFP04] Alexandra Mendes, Joa£o Ferreira, and JoeÃ© ProcnÃ§a. Camila prelude. Technical report, September 2004. http://wiki.di.uminho.pt/wiki/bin/view/Main/AlexandraMendes/CPrelude.pdf.

[Muk97] Paul Mukherjee. Automatic translation of vdm-sl specifications into gofer. In *FME '97: Proceedings of the 4th International Symposium of Formal Methods Europe on Industrial Applications and Strengthened Foundations of Formal Methods*, pages 258–277. Springer-Verlag, 1997.

[Nec05] C. Necco. Polytypic data processing. Master's thesis, Facultad de Cs. Físico Matemáticas y Naturales,University of San Luis, Argentina, 2005. (Submitted.).

[Oli97] José N. Oliveira. Why formal methods? http://www.di.uminho.pt/ jno/html/cam-wfm.html, 1997.

[Oli01] José N. Oliveira. *A Quick Look at Monads*. 2001.

[PSSC04] André Pang, Don Stewart, Sean Seefried, and Manuel M. T. Chakravarty. Plugging haskell in. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 10–21. ACM Press, 2004.

[Syl] Sebastian Sylvan. Why does haskell matter? http://www.haskell.org/complex/why_does_haskell_matter.html.

[TM92] J. L. Turner and T. L. McCluskey. *The Construction of Formal Specifications: An Introduction to the Model-Based and Algebraic Approaches*. October 1992.

[vdm] Vdm: Vienna development method - faq. http://www.vienna.cc/evdm.htm.

[Vie93] Robert L. Vienneau. A review of formal methods. Technical report, May 1993. http://www.dacs.dtic.mil/techs/fmreview.

[Vis96] Joost M.W. Visser. Evolving algebras. Master's thesis, August 1996.

[Wad90] Philip Wadler. Comprehending monads. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78. ACM Press, 1990.

[Wad92] Philip Wadler. The essence of functional programming. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14. ACM Press, 1992.

A Haskell examples

A.1 Stack Algebra

```
{-# OPTIONS -fglasgow-exts  #-}
module Camila.Examples.StackAlg where

import Camila.Prelude

-- Datatype definition
type Stack = [Int]

-- Datatype invariant
instance CData Stack where
   inv s = (filter odd s) == s

-- Functions
init' :: Stack
init' = []

empty :: Stack -> CamilaError Bool
empty s = do inv' s
             return (s == [])

push :: Int -> Stack -> CamilaError Stack
push n s = do s1 <- Right ([n]++s)
              inv' s1

-- First, we define a local datatype
data Stack_pop = Stack_pop Stack deriving Show

-- Second, we define it as instance of Partial
instance Partial Stack_pop where
   pre (Stack_pop s) = not $ runError $ empty s

-- Pop is defined as follows:
pop :: Stack -> CamilaError Stack
pop s = do inv' s
           pre' (Stack_pop s)
           return (tail s)


data Stack_top = Stack_top Stack deriving Show

instance Partial Stack_top where
   pre (Stack_top s) = not $ runError $ s == []

top :: Stack -> CamilaError Int
top s = do inv' s
           pre' (Stack_top s)
```

```
          return (head s)


data Stack_add = Stack_add Stack deriving Show

instance Partial Stack_add where
    pre (Stack_add s) = length s >= 2

add :: Stack -> CamilaError Stack
add s = do inv’ s
           pre’ (Stack_add s)
           ret <- Right $ [s!!0 + s!!1] ++ tail (tail s)
           return ret

-- Examples:

alg_prog1 = do s <- push 1 []
               s1 <- push 2 s -- invariant violation
               s2 <- pop s1
               return s2

alg_prog2 = do s <- push 1 []
               s1 <- pop s
               s2 <- pop s1 -- pre-condition violated
               return s2

alg_prog3 = do s1 <- push 1 []
               s2 <- push 3 s1
               return s2
```

A.2 Stack Object

```
{-# OPTIONS -fglasgow-exts #-}
module Camila.Examples.StackObj where

import Camila.Prelude
import Camila.Examples.StackAlg
import AltData.Typeable

-- instance variables
data StackIVars = StackIVars { stateStack :: Stack } deriving Show
-- initial state
initialStack = StackIVars { stateStack = init’ }

-- We have to define StackIVars as instance of AltData.Typeable,
-- to allow the interpreter to make dynamic values
instance Typeable StackIVars where
    typeOf _ = mkAppTy (mkTyCon "StackIVars") []
```

```
-- operations
opclear :: ST StackIVars ()
opclear = modify (\s -> s { stateStack = init' })

oppush :: Int -> ST StackIVars ()
oppush a = do actual <- gets stateStack
              --liftE $ inv' actual -- push verifies it
              new <- liftE $ push a actual
              modify (\s -> s { stateStack = new})

oppop :: ST StackIVars Int
oppop = do actual <- gets stateStack
           t <- liftE $ top actual
           new <- liftE $ pop actual
           modify (\s -> s { stateStack = new })
           return t

optop :: ST StackIVars Int
optop = do actual <- gets stateStack
           element <- liftE $ top actual
           return element

-- Examples (of state + error)
obj_prog1 = do opclear
               oppush 1
               oppush 3
               oppush 5
               i <- optop
               liftIO $ putStrLn $ "Top: " ++ show i
               stack <- gets stateStack
               liftIO $ putStrLn $ "Stack: " ++ show stack

obj_prog2 = do opclear
               oppush 1
               oppush 3
               oppush 2
               i <- optop
               liftIO $ putStrLn $ "Top: " ++ show i
               stack <- gets stateStack
               liftIO $ putStrLn $ "Stack: " ++ show stack

obj_prog3 = do opclear
               oppop


--- Functions needed to animate the examples
exec :: (Show a) => ST StackIVars a -> IO ()
exec p = do result <- run p
            if (isLeft result) then putStrLn $ "ERROR: " ++ reason (unLeft result)
                               else putStrLn $ show $ unRight result
```

```
run p = runInstance p initialStack

unLeft (Left a) = a
unRight (Right a) = a

isRight (Right _) = True
isRight _ = False

isLeft (Left _) = True
isLeft _ = False
```