Generic Libraries in C++ with Concepts from High-Level Domain Descriptions in Haskell

A Domain-Specific Library for Computational Vulnerability Assessment

Daniel Lincke[1], Patrik Jansson[2], Marcin Zalewski[2], and Cezar Ionescu[1]

[1] Potsdam Institute for Climate Impact Research, Potsdam, Germany
{daniel.lincke,ionescu}@pik-potsdam.de
[2] Chalmers University of Technology & University of Gothenburg, Gothenburg, Sweden
{patrikj,zalewski}@chalmers.se

Abstract. A class of closely related problems, a problem domain, can often be described by a domain-specific language, which consists of algorithms and combinators useful for solving that particular class of problems. Such a language can be of two kinds: it can form a new language or it can be embedded as a sublanguage in an existing one. We describe an embedded DSL in the form of a library which extends a general purpose language. Our domain is that of vulnerability assessment in the context of climate change, formally described at the Potsdam Institute for Climate Impact Research. The domain is described using Haskell, yielding a domain specific sublanguage of Haskell that can be used for prototyping of implementations.

In this paper we present a generic C++ library that implements a domain-specific language for vulnerability assessment, based on the formal Haskell description. The library rests upon and implements only a few notions, most importantly, that of a monadic system, a crucial part in the vulnerability assessment formalisation. We describe the Haskell description of monadic systems and we show our mapping of the description to generic C++ components. Our library heavily relies on *concepts*, a C++ feature supporting generic programming: a conceptual framework forms the domain-specific type system of our library. By using functions, parametrised types and concepts from our conceptual framework, we represent the combinators and algorithms of the domain. Furthermore, we discuss what makes our library a domain specific language and how our domain-specific library scheme can be used for other domains (concerning language design, software design, and implementation techniques).

1 Introduction

The road from a domain of expert knowledge to an efficient implementation is long and perilous, often resulting in an implementation that bears little apparent resemblance to the domain. Still, the practitioners of high-performance computing are willing to pay the price of overwhelmingly low-level implementations to guarantee performance. In the current day and age, however, it is possible to increase the level of abstraction by applying *generic programming* techniques to provide a high-level library for a domain that allows structured use of existing, efficient code. In this paper, in particular,

W.M. Taha (Ed.): DSL 2009, LNCS 5658, pp. 236–261, 2009.

we show how to first describe a domain in a high-level *functional* style, close to mathematical notation but executable, and, then, transform this description into a strongly generic library that corresponds almost directly to the initial description. Specifically, we present the process by which we derived a generic library for vulnerability assessment (in the context of environmental sciences) in C++ (ANS, 2003; Stroustrup, 1997).

The concept of "vulnerability" plays an important role in many research fields, including for instance climate change research. As we will discuss in section 2, there is no unique definition of this term. However, there is a simple common structure underlying the definitions in the usage of "vulnerability", which can be shortly described as measuring the possible harm for a given system.

Our C++ implementation is directly based on the high-level description given in Haskell (Peyton Jones, 2003) by Ionescu (2009). Ionescu relies on functional features of Haskell, such as function types and type constructors, to represent the underlying category-theoretical notions of, among others, functors, coalgebras, monads, and, crucially, of a construct of *monadic dynamical systems*, which we will introduce and explain in section 2.2. While the description can be actually executed, its design is not focused on efficiency. Furthermore, it not generic enough to be applied in a practical setting of existing implementations, tool sets, and so on.

Our C++ library maintains the abstract, high-level correspondence with the domain notions introduced by Ionescu but it makes efficient implementations possible by providing a generic interface for use of high-performance components. We are able to achieve the abstraction by basing the library on a hierarchy of *concepts*, a language mechanism in C++ supporting generic programming (Gregor et al., 2006, 2008c)[1]. While the use of concepts is not novel itself (e.g., Gregor et al., 2008b), we use a conceptual framework to generically specify "types of types" corresponding to the types from the domain description. The library consists of the three levels:

- a conceptual framework specifying concepts (type classes) for functions, functors, monads, and so on;
- parametrised types, themselves "typed" by concepts, that represent domain constructs and combinators;
- and, finally, parametrised functions representing the algorithms applicable in a given domain.

The library can be seen as a domain specific language embedded into C++, complete with its own type system and language constructs.

A library like ours is much more complex and verbose than its high-level specification in Haskell but it is surprisingly similar and nimble from the point of view of the user. For example, vulnerability is cleanly specified in Haskell as a function of the following type (see Sect. 2 for details):

[1] Haskell provides a similar mechanism of *type classes* (Wadler and Blott, 1989) that could potentially serve as the basis for a Haskell library similar to our C++ library. For comparison of Haskell type classes and C++ concepts, see the work of Bernardy et al. (2008).

```
vulnerability :: (Functor m, Monad m) =>
                 (i -> x -> m x) -> ([x] -> harm) ->
                 (m harm -> v) -> [i] -> x -> v
```

In the C++ library, the type of the function has to be translated to the conceptual framework (see Sects. 3 and 4 for details):

```
template <MonadicSystem Sys, Arrow Harm, Arrow Meas, ConstructedType Inputs>
requires ConstructedType<Harm::Domain>, ConstructedType<Meas::Domain>,
  SameType<Harm::Domain::Inner, Sys::Codomain::Domain>,
  SameType<Meas::Domain::Inner, Harm::Codomain>,
  SameTypeConstructor<Meas::Domain, Sys::Codomain::Codomain>,
  SameType<Inputs::Inner, Sys::Domain>,
  FMappable<Harm, Rebind<Sys::Codomain::Codomain, Harm::Domain>::Computed>;

Meas::Codomain
vulnerability_sys(Sys, Harm, Meas, Inputs, Sys::Codomain::Domain);
```

In C++, the type of the Haskell `vulnerability` function must be expressed in terms of concepts that form the basis of the library (see Sect. 5 for discussion of the translation scheme). While clearly more complex from the point of view of a library developer, the complexity is hidden when the function is used:

```
vulnerability_sys(my_sys, my_harm, my_measure, my_inputs, my_state);
```

The user must declare the necessary *concept maps* that describe how the types to which `vulnerability_sys` is applied model the necessary concepts. The mapping through concept maps makes it possible to use the library with the existing high-performance implementations, keeping the complexity manageable, since the necessary concept maps are specified one at a time, and with the right use of C++ concept features many concept maps can be generated by the compiler.

In order to give an overview over the library, the core concepts from the library are outlined in Fig. 1. In the left column we give the signature of the C++ concept and refinement relations. On the right side we show how the parameters of the concept match the Haskell signature (we write `C` $\sim$ `t` for "C++ type `C` corresponds to Haskell type `t`"). Furthermore we give a brief explanation of what the concept is intended to model. The concepts are divided into a few groups. This grouping does not imply any hierarchy of the used concepts, it is only for representation reasons.

In the remainder of the paper, we outline the progression from the domain to an efficient implementation, illustrated by our particular path from the domain of vulnerability assessment to our C++ library. In Sect. 2, we present the concept of vulnerability as used in the climate change community and its mathematical formalisation in Haskell. Section 3 shows the C++ implementation of the domain specific constructs (concepts, parametrised types and functions) and Sect. 4 shows how the library can be used.

Arrow types	
`Arrow<class Arr>`	`Arr` ∼ `a -> b`: describes general function types
`CurriedArrow<class CArr> : Arrow <CArr>`	`CArr` ∼ `a -> (b -> c)`: functions whose codomain is another function
`Coalgebra<class Arr> : Arrow<Arr>`	`Arr` ∼ `x -> f x`: functions whose codomain is a functor application
`CoalgebraWithInput<class Arr> : Arrow<Arr>`	`Arr` ∼ `(a,x)-> f x`: coalgebras with an additional input
`MonadicCoalgebra<class Arr> : Coalgebra<Arr>`	`Arr` ∼ `x -> m x`: functions whose codomain is a monad application
`MonadicCoalgebraWithInput<Arr> : CoalgebraWithInput<Arr>`	`Arr` ∼ `(a,x)-> m x`: monadic coalgebras with an additional input
Type constructors	
`ConstructedType<class T>`	`T` ∼ `f x`: the type `T` is a constructor application
`SameTypeConstr<class T1, class T2>`	`T1` ∼ `f x`, `T2` ∼ `f y`: two constructed types have the same type constructor
Functors and Monads	
`FMappable<class Arr, class Type>`	`Arr` ∼ `a -> b`, `Type` ∼ `f a`: provides operation `FMap` for the function `Arr` and the type `Type`
`MBindable<class MX, class Arr>`	`MX` ∼ `m x`, `Arr` ∼ `x -> m y`: provides operation `MBind` for the type `MX` and the function `Arr`
`MReturnable<class MX>`	`MX` ∼ `m x`: provides operation `MReturn` for `MX`
Miscellaneous	
`Monoid<class Op>`	`Op` ∼ `t -> t -> t`: `Op` is a monoid operation
Monadic system	
`MonadicSystem<class Sys> : CurriedArrow<Sys>`	`Sys` ∼ `t -> (x -> m x)`: `Sys` is a monadic system

Fig. 1. Overview of the C++ concepts in the library

The process by which we arrived at C++ code is highly reusable: while some human ingenuity is required much of the work is mechanical, translating a mathematical description into a generic library. Furthermore, some parts of our library, such as generic representation of function types and type constructors, can be directly reused when implementing other domains. In fact, in Sect. 5 we present a scheme for translation of functional Haskell types into our reusable, "types of types" concepts. Yet, although our library is generic, we still made some particular design choices. In Sect. 6 we survey related work and in Sect. 7 we discuss some of the choices and possible alternatives, including the ideas on how to further generalise the C++ representation.

1.1 Preliminaries

We use Haskell code for describing the mathematical model and C++ code for describing the implementation. Here we give a very short description of some of the notation used. It is worth noting already from the start that `::` is used in two different ways: in Haskell it is the "has type" keyword, while in C++ it is the "scope resolution" operator.

Haskell. For specification of the mathematical model we use the Haskell programming language (Peyton Jones, 2003), without any of the language extensions. We make

frequent use of the type class `Monad m` with members `return` and `(>>=)` (pronounced "bind") and the class `Functor f` with the member `fmap`.

```
class Functor f where
  fmap   ::  (a -> b) -> f a -> f b
class Monad m where
  (>>=)  ::  m a -> (a -> m b) -> m b
  return ::  a -> m a
```

For Kleisli composition we use `(<=<) :: (b -> m c) -> (a -> m b) -> (a -> m c)` and normal function composition is `(.) :: (b -> c) -> (a -> b) -> (a -> c)`.

We use two recursion operators for lists — `foldr` and `scanr` — which are best introduced by example. If we define `sum = foldr (+) 0` and `sums = scanr (+) 0` then `sum [3,2,1]` is `6` and `sums [3,2,1]` are the partial sums `[6,3,1,0]`. Following a similar pattern we use `foldr` to compose lists of monadic actions:

```
compose  :: Monad m => [x -> m x] -> (x -> m x)
compose  = foldr (<=<) return
```

For testing we write predicates (parametrised test cases) as normal Haskell functions run by QuickCheck Claessen and Hughes (2000). We use the type `MyFloat` (a wrapper around a normal `Float`) to enable "approximate equality" (needed for testing floating point computations). We use the type constructor `SimpleProb` for probability distributions with finite support.

C++. In C++, genericity is achieved through the use of templates (Vandervoorde and Josuttis, 2002), which are pieces of code parametrised by types and values. Type parameters of templates are constrained by concepts and implementations are declared to model particular concepts in concept maps; when templates are instantiated, concept maps are used to bind names dependent on type parameters to the implementations provided by the user (Gregor et al., 2006, 2008c).

A concept can be seen as a *predicate* (or relation) over types. When a type (or a tuple of types) satisfies this predicate, we say that it forms (or they form) a *model* of the concept. The body of a concept can specify a number of *associated entities*: values, functions, types and axioms. Definitions for these entities are provided separately, in concept maps, for all models of the concept. The following snippet of code is a complete example of using concepts:

```
1 concept LessThanComparable<typename T> { bool operator<(T x, T y); }
2
3 template<typename T>  requires LessThanComparable<T>
4 T min(T x, T y) { return x < y ? x : y; }
5
6 concept_map LessThanComparable<int> { }
7
8 int x = min(2, 3);
```

The concept `LessThanComparable`, defined on Line 1, defines a predicate on a single type and requires an associated operator `<`. The concept is then used, on Line 3, to constrain the sole type parameter of the `min` algorithm, making the values of the parameter type comparable with the operator `<`. The built-in type **`int`** is declared to model `LessThanComparable`, on Line 6. The definition of the associated operator is not given explicitly. In such situations, the compiler is required to attempt to find a matching definition in the context, and, in this case, it finds the built-in operator for **`int`**. Finally, the usage of concept-constrained `min` is demonstrated on Line 8: the application `min(2, 3)` concept-checks, since the required concept map has been defined.

To use concepts in generic code fragments the keyword **`requires`** can be used to specify the requirements of template parameters. Unary concept requirements can be specified directly in the template parameters, that is

```
template<class A>  requires  Concept_B<A> ...
```

can be written as **`template`**`<Concept_B A> ...` and we frequently use this notation.

In C++ we do not use an extra data type for real numbers. We use floating point types and if we need "approximate equality" we use an overloaded operator `==`. All the C++ code examples in this paper were implemented and tested with the ConceptGCC compiler[2] (version alpha 7). It is based on the gcc compiler, version 4.3.

Furthermore, in all C++ code we omit unnecessary details such as **`const`** qualifiers or reference (`&`) type modifiers — while these are important for performance they do not contribute to the exposition. Finally, in template parameter lists, the keywords **`class`** and **`typename`** are freely exchangeable — the variations in the paper are motivated by space constraints.

2 Vulnerability Modelling

In the past decade, the concept of "vulnerability" has played an important role in the fields such as climate change, food security, or natural hazard studies. Vulnerability studies have been useful in alerting policymakers to the possibility of precarious situations in the future.

However, definitions of vulnerability vary: there seem to be almost as many definitions of vulnerability as case studies conducted. Wolf et al. (2008), for instance, analyse the usage of 20 definitions of vulnerability. An allegory frequently used to describe the terminology is the Babylonian confusion, and the need for a common understanding has repeatedly been expressed in the literature (Brooks, 2003; Janssen and Ostrom, 2006).

While definitions of vulnerability are usually imprecise, Ionescu (2009) proposes a mathematically defined framework that underlies the definitions and the usage of the term *vulnerability*. Ionescu's framework applies to those studies that "project future conditions" by using computational tools and that can thus be designated as computational vulnerability assessment. This common framework for vulnerability assessment consist of three primitive concepts: an *entity* (which is subject to possible future

[2] `http://www.generic-programming.org/software/ConceptGCC/`

harm), *possible future evolutions* (of the entity), and *harm*. Next, we briefly summarise Ionescu's development. First, we discuss the primitive concepts and, then, their extension to *monadic dynamical systems*.

2.1 A Simple Mathematical Model

An entity, its current situation, and its environment are described by a state. In vulnerability computations, the current state of an entity is only a starting point; it is the *evolution* of the state over time that is the main focus of interest. A single evolution path of an entity is called a *trajectory*. In a first, simple vulnerability model, the evolution of an entity is computed in a single step:

```
possible  :: State -> F Evolution
```

The `possible` function maps a state into an `F`-structure of future evolutions. Two types and one type constructor are involved: `State` is the type of the state of an entity, `Evolution` is the type of the evolution of an entity, and `F` is the type constructor representing the structure of evolutions (e.g., a set). In this simple model we take `Evolution = [State]`: an evolution is a finite sequence of the states that an entity successively assumes over time. This notion is sufficient, as most vulnerability assessments are made for a given horizon, which is expressible as a finite evolution. The constructor `F` is a functor (see Sect. 1.1) that describes the nature of the future computation. For example, if the future of an entity is deterministic, one would have `F = Id`, the identity functor, giving a single evolution. Sometimes, a more sophisticated model may be necessary. For instance, a stochastic model might be used, which computes probability distributions over trajectories; the stochastic functor introduced in Sect. 1.1 captures such situations (`F = SimpleProb`). Some other models might produce just lists of trajectories, without assigning a specific probability to each possibility. For these models, the trajectories are enclosed in the list functor (`F = []`).

The second ingredient of a vulnerability computation is a harm judgement function:

```
harm      :: Evolution -> Harm
```

The `harm` function maps evolutions into values of type `Harm`, which describes the nature of the harm assessment. One of the simplest harm assessments is a threshold measurement. Examples of thresholds that can be crossed include temperature that grows too high or a budget that is too low. Correspondingly, threshold harm measurement judges two possibilities, namely "yes, there is harm for this evolution" or "no, there isn't any harm when the entity evolves in this way." In such cases, harm is simply a boolean value (`Harm = Bool`). In other models, the harm value for each trajectory may be measured as a real value (`Harm =` $\mathbb{R}$). A more sophisticated model might measure multiple dimensions of harm, all expressed as real values, which would lead to `Harm =` $\mathbb{R}^N$.

The `harm` function measures the harm of a single evolution, but a model produces an `F`-structure of evolutions, given some implementation of the function `possible`. The harm function must then be mapped to the evolutions, using the `map` function of the functor `F` (see Sect. 1.1). A measure function takes the resulting `F`-structure of harm values and summarises them into an overall measurement:

```
measure   :: F Harm -> V
```

The codomain of the `measure` function could be, for example, the real numbers (to get the result as a single number, a way of expression policymakers prefer) or a multidimensional space (to express various dimensions of vulnerability instead of collapsing it into a single number). Common examples for the measure function are, for instance, `measure = maximum` in the case of `F = []` and `Harm = Bool`, or `measure = expectation` in the case of `F = SimpleProb` and `Harm =` $\mathbb{R}$.

Combining these three functions gives the vulnerability computation:

```
vulnerability :: State -> V
vulnerability =  measure . fmap harm . possible
```

This computation captures the idea behind many vulnerability definitions: we *measure* the *harm* that might be *possible*. Ionescu (2009) specifies certain conditions that the ingredients of the vulnerability computation must fulfil; the measure function, for example, has to fulfil a monotonicity condition.

The following example illustrates the use of the vulnerability computation framework introduced in this section:

```
1  type Economy_State = (Float, Float,     Float, ...)
2                       -- GDP,  GDP-growth, Unemployment Rate, more ...
3  type Economy_Evolution = [Economy_State]
4
5  possible_economy :: Economy_State -> [ Economy_Evolution ]
6  possible_economy x = -- ... implement a model of an economy here
7
8  economic_harm :: Economy_Evolution -> Float
9  economic_harm ec_ev =
10   -- ... return 0 if GDP-growth is always >= 0,
11   --     otherwise the absolute value of the biggest GDP-loss
12
13 economic_vulnerability :: Economy_State -> Float
14 economic_vulnerability  =  maximum . (fmap economic_harm) . possible_economy
```

The implementation of the economy evolution function is unspecified, but the type indicates that it is performed non-deterministically. Harm is defined as the greatest decrease in Gross Domestic Product (GDP) over a trajectory; the harm function returns zero if the GDP is always increasing. In the final vulnerability computation, the `maximum` function is used as measure, defining vulnerability as the greatest economic loss over all possible evolutions of the economy in a given state.

2.2 Monadic Systems and Vulnerability

The simple model of vulnerability presented in the previous section has two main disadvantages. First, it is not very generic: as shown in the economy example we have to write a new function for every vulnerability computation. This can easily be fixed by making `possible`, `harm` and `measure` inputs to the vulnerability computation. Second, the model is inflexible: the evolution function of the simple model only consumes the initial state of a system as the input. In practise, however, an evolution function often has more inputs. In particular, an evolution may depend on auxiliary control arguments that are not part of the initial state of a system.

Ionescu (2009) explores possible structures which might provide the required flexibility. One the one hand the multitude of systems which might be involved has to be represented, on the other hand we want an easy way of computing iterations. It turns out there, that a functorial structure on the possible evolutions is not suitable for the trajectory computations and that the use of a monad instead of a functor gives the possibility to iterate a system. Therefore, the notion of a *monadic dynamical systems* (MDS) was introduced. In contrast to the `possible` function, trajectories of a model defined by an MDS can be computed in multiple separate steps. For example, trajectory of an MDS can be computed up to a certain point, the result archived, and the computation continued later with this intermediate result as a starting point. An MDS is a mapping with the signature `T -> (X -> M X)`, where `M` is a monad (see Sect. 1.1). An MDS takes a value of type `T` (e.g., `T = Nat`) to a *(monadic) transition function* of type `X -> M X`. Ionescu requires some further conditions, but we don't need them for this paper.

An MDS need not be directly defined by the user. Often it is built by constructor functions, given a transition function. In the simplest case, the function describes only the transition from one state into the possible next states. The following function defines for instance a non-deterministic transition (with state type `Int` and the list monad):

```
1 my_discrete_trans :: Int -> [Int]
2 my_discrete_trans x = [x-1, x+1]
```

We can construct an MDS from `my_discrete_trans` using `discrete_sys`:

```
1 type Nat = Int
2 discrete_sys :: (Monad m)  =>  (x -> m x) -> (Nat -> x -> m x)
3 discrete_sys f 0        =  return
4 discrete_sys f (n + 1)  =  (>>= f) . discrete_sys f n
```

Not all transitions of interest are that simple. Often, a transition function does not only map a state into an `M`-structure of states, but also takes an additional control parameter. This parameter describes some external input which is not a part of the state. Such function describes a family of transition functions, one for each control input. Consider the following example:

```
my_input_trans :: MyFloat -> (MyFloat -> SimpleProb MyFloat)
my_input_trans delta x = SP [(x + delta, 0.75),
                             (x - delta, 0.25)]
```

Here, every input `delta` gives a different transition function. The function itself is not an MDS (because it fails to satisfy the MDS laws). But it can be used to construct a monadic system by using the following constructor:

```
input_system :: (Monad m)  =>  (i  -> x -> m x) -> ([i] -> x -> m x)
input_system trans = compose . map trans
```

Furthermore, given an MDS, two iteration functions can be defined: the macro trajectory function, `macro_trj`, computes the evolution of an `M`-structure of states, and the micro trajectory, `micro_trj`, computes the `M`-structure of evolutions (sequences of states). Both computations start with a sequence of inputs of type `i` and an initial state. The macro trajectory describes all possible states at each step of the computation, but does not connect them:

```
macro_trj :: (Monad m) =>  (i -> (x -> m x)) -> [i] -> x -> [m x]
macro_trj sys ts x  =  scanr (\i mx -> mx >>= sys i) (return x) ts
```

The micro trajectory records the single evolution paths including all intermediate states:

```
micro_trj :: (Monad m) =>  (i -> (x -> m x)) -> [i] -> x -> m [x]
micro_trj sys ts x  =  compose (map (addHist . sys) ts) [x]

addHist :: (Monad f) => (x -> f x) -> [x] -> f [x]
addHist g (x : xs) = liftM (:(x : xs)) (g x)
```

Computing the trajectories for the MDS constructed out of `my_discrete_trans` from the initial state `myx = 0` and the input sequence `myts = [1,1,1]` yields:

```
  macro_trj (discrete_sys my_discrete_trans) myts myx ==
  [ [-3,-1,-1,1,-1,1,1,3],
    [-2,0,0,2],
    [-1,1],
    [0]]
  micro_trj (discrete_sys my_discrete_trans) myts myx ==
  [ [-3,-2,-1, 0], [-1,-2,-1, 0],
    [-1, 0,-1, 0], [ 1, 0,-1, 0],
    [-1, 0, 1, 0], [ 1, 0, 1, 0],
    [ 1, 2, 1, 0], [ 3, 2, 1, 0]
  ]
```

The example clarifies the nature of the two trajectory computations: The macro trajectory provides the information that the state of this system is 0 at the beginning. After one step the state is either -1 or 1, after two steps it can be in state -2, 0 or 2 and so on. The micro trajectory contains all single trajectories the system can take, from `[-3,-2-1,0]`, where we always take -1 path, to `[3,2,1,0]`. Note, that the computed states are prepended to the list, leading to the unintuitive ordering of states with the final state at the head of the list. The input of `myts` are processed in the same order: starting from the end of the list and proceeding to the first element.

Many computational models used in the climate change community already implicitly implement the structure of an MDS. For instance, the climate model CLIMBER (Petoukhov et al., 2000) has an internal state which represents the current configuration of certain physical parameters. In addition, CLIMBER takes as an input the concentration of greenhouse gases. The CLIMBER model can be described as an MDS built from a family of transition functions with signature `climber :: GHG -> (Climber_st -> Id Climber_st)`, where `GHG` is the type of values representing concentration of greenhouse gases, and evolution uses the `Id` monad, since CLIMBER is deterministic.

To tie back to vulnerability assessment, the `micro_trj` computation for an MDS fits the signature of the `possible` function in a vulnerability computation. Thus, the `possible` can be computed as the structure of micro-trajectories of an MDS. Taking this into account we can compute vulnerability as:

```
1 vulnerability' :: (Functor m, Monad m) =>
2                   (i -> x -> m x) -> ([x] -> harm) ->
3                   (m harm -> v) -> [i] -> x -> v
4 vulnerability' sys harm measure ts = measure . fmap harm . micro_trj sys ts
```

Using the above definitions, the examples of the `possible` functions from the previous section, with `F = Id`, `F = []`, or `F = SimpleProb` as the resulting structures, can all be easily translated to an MDS, since all these functors are also monads.

3 C++ Implementation of the Vulnerability Model

The Haskell description of vulnerability which has been provided in the previous section was rather on a mathematical level than a computational library. Due to its laziness and its clean notation Haskell is useful for modelling and prototyping language, which enables one to write operational definitions that can be executed.

As seen in the previous section, the mathematical description of vulnerability and monadic systems is a generic one, for instance a monadic system has three type parameters involved. To take advantage of this flexibility we implemented generic Haskell software components. If we want to translate these description into a C++ library, we have to use generic programming techniques, which are nowadays well supported (by the C++ concepts language extension).

However, Haskell is not the first choice for practitioners of high-performance computing when implementing generic libraries. In Haskell there are overheads in terms

of the runtime system, garbage collection, dictionary passing etc. Many of these problems can be overcome by tuning the GHC compiler to do clever optimisations and specialisations, but in most production environments there is a lack of Haskell expertise. Furthermore C++ has many existing scientific computing libraries which can be used to implement computations in transition functions for monadic systems. In addition C++ is widely distributed and supports multiple software development paradigms on different level of abstraction. Therefore C++ was chosen as the implementation language.

As shown by Bernardy et al. (2008) there are many similarities between generic programming in Haskell (using type classes) and C++ (using concepts). But there are some important differences between these two languages which make a direct translation of the Haskell model from the previous section difficult. Two of the differences are:

- Haskell offers a built in type for functions, C++ does not ;
- Type constructors (parametrised types) are well supported in Haskell, one can use them as parameters in generic functions or combine them. Type constructors (template classes) in C++ do not offer this first-class support.

Due to these problems, we have to implement a conceptual framework specifying a system of "types of types"

3.1 The Conceptual Framework

We use concepts to specify types which are not part of standard C++, such as functions, functors and monads. Function types, for instance, are encoded by the following `Arrow`[3] concept, where we hint at the relation to Haskell types with a comment:

```
1 // F  ~  a -> b
2 concept Arrow<class F> {
3   typename Domain;
4   typename Codomain;
5
6   Codomain operator () (F, Domain);
7 };
```

This concept is related to the design pattern of function objects. A function object is simply any object that can be called as if it is a function, thus providing a application operator. In that sense, the concept `Arrow` provides a static interface for function objects. Its name comes from the fact that we do not state that these types model functions in the mathematical sense, for instance side effects might be involved. Furthermore the concept framework provides additional concepts for special kinds of arrows, e.g. the concept `CurriedArrow` describing mappings of type `f :: A -> (B -> C)`.

Another important issue is how to describe types which are constructed by explicitly applying a type constructor. We will call these types constructed types. As we cannot (due to the technical limitations) describe type constructors directly, the concept `ConstructedType` is used to describe these types. An associated type is used to keep the original type:

[3] This concept does not correspond to the Haskell type class `Arrow (a :: *->*->*)`.

```
// T  ~  g Inner
concept ConstructedType<typename T> {
  typename Inner;
};
```

A parametrised concept mapping can be used to declare all STL-containers to be constructed types and therefore instances of this concept:

```
template<std::Container C>
concept_map ConstructedType<C> {
  typedef C::value_type Inner;
};
```

Furthermore there are concepts modelling relations between types and/or type constructors: the built-in concept `SameType` decides if two types are equal and the concept `SameTypeConstructor` is fulfilled for all pairs of constructed types sharing the same type constructor.

Based on these rather simple concepts we can model more complex concepts. For instance, we implement the `MBindable` concept as a relation between types:

```
//   MX  ~  m x;    Arr  ~  x -> m y
concept MBindable<class MX, class Arr> {
  requires ConstructedType<MX>, Arrow<Arr>,
           ConstructedType<Arr::Codomain>,
           SameType<MX::Inner,Arr::Domain>,
           SameTypeConstructor<MX,Arr::Codomain>,
           CopyConstructible<Arr::Codomain>;

  Arr::Codomain MBind(MX, Arr);
};
```

It states that there is an `MBind` operation for a pair of types `MX` and `Arr` if certain conditions (stated by requirements) are fulfilled. This member operation implements the monadic bind. In Haskell the bind operation is associated with the monadic type constructor, but here bind is associated with the types of its arguments. A similar concept `MReturnable` models types supporting `mreturn` and `FMappable` is used to model that there is an `fmap` operation.

Using `FMappable` we can describe *coalgebras*: arrows of type `X -> F X` where `F` is a functor. Similarly, using `MBindable` and `MReturnable` we can describe *monadic coalgebras*. We implemented concepts for both of them, the `MonadicCoalgebra` for instance looks as follows:

```
// Arr  ~  x -> m x
concept MonadicCoalgebra<class Arr>  :  Coalgebra<Arr> {
  requires MBindable<Codomain, Arr>, MReturnable<Codomain>;
};
```

Moreover, we provide a concept `MonadicCoalgebraWithInput` which describes arrows of type `f :: (A, X) -> M X` where `M` is a monad. This arrow type is necessary to implement monadic systems with an external control (realised as an input to the system).

On the top of the hierarchy of concepts we specify the main concept of our type specification system: monadic systems. As monadic systems are defined as a special form of a curried mapping, we refine the concept `CurriedArrow`:

```
// Sys  ~  t -> (x -> m x)
concept MonadicSystem<class Sys> : CurriedArrow<Sys> {
  typename Op; // Op  ~  (t, t) -> t
  requires Monoid<Op>, MonadicCoalgebra<Codomain>;
};
```

Note that there are more conditions, in particular we have the compatibility conditions for the mapping, C++ concepts offer the possibility to express them as axioms which can be embedded concepts. The expression and processing of axioms is beyond the scope of this paper.

3.2 The Type System

The concepts presented so far build the type specification system of the language. However the language does not only contain abstractions of datatypes in the form of concepts, we also provide specific constructors realised as parametrised types. These types are on two levels: constructs which support the language by implementing helpful operations and constructs which are part of the domain. Helper constructs are for instance function operations like arrow composition, arrow pairing and arrow currying. The operation `kleisli_compose` used in the description of the second axiom above implements a special composition of monadic coalgebras and is therefore also a helper construct. Examples for domain specific constructs are constructors for monadic systems, e.g. a constructor for a monadic system with input, which is similar to the `input_system` constructor we have in the Haskell implementation:

```
// Trans  ~  (a,x) -> m x;  Cont  ~  [a]
template<MonadicCoalgebraWithInput Trans, ConstructedType Cont>
requires SameType<Trans::Domain::First, Cont::Inner>
class Monadic_System_Input {
public:
  typedef Cont                            Domain;
  typedef InpMonadicSysCodom<Trans,Cont> Codomain;
  typedef Concatenation<Cont>             Op1;

  Monadic_System_Input(Trans trans) : my_trans(trans) {}

  Codomain operator() (Domain l) {
    InpMonadicSysCodom<Trans,Cont> ret(my_trans,l);
    return ret;
  }

private:
  Trans my_trans;
};
```

The codomain of this monadic system is a class `InpMonadicSysCodom` representing a monadic coalgebra of type `X -> MX` which is constructed out of a function of type `(T,X)-> MX` and a sequence of type `[T]`. The type `Monadic_System_Input` models the concept of a monadic system. That is made explicit in the code by a **`concept_map`**, but we omit it as it adds no new information.

3.3 Algorithms

The library contains generic functions which implement the algorithms developed in the formalisation, including the two trajectory computations from Sect. 2. For example, the `macro_trj` function is defined as follows:

```
macro_trj :: (Monad m) =>  (i -> (x -> m x)) -> [i] -> x -> [m x]
macro_trj sys ts x  =  scanr (\i mx -> mx >>= sys i) (return x) ts
```

This function translates to the following C++ implementation:

```
template <MonadicSystem Mon_Sys, BasicOutputIterator Out_Iter>
requires SameType<Mon_Sys::Codomain::Codomain, Out_Iter::value_type>
void macro_trj_init(Mon_Sys sys, Out_Iter ret, Mon_Sys::Codomain::Domain x)
{
  typedef Mon_Sys::Codomain::Codomain MacroState; // MacroState  ~  m x
  MacroState mx = mreturn(x);
  *ret = mx; ++ret;
}

template <MonadicSystem Mon_Sys, ForwardIterator Inp_Iter,
          BasicOutputIterator Out_Iter>
requires SameType<Inp_Iter::value_type, Mon_Sys::Domain>,
         SameType<Out_Iter::value_type, Mon_Sys::Codomain::Codomain>
void macro_trj(Mon_Sys sys, Inp_Iter controls_bg, Inp_Iter controls_end,
               Out_Iter ret, Mon_Sys::Codomain::Codomain mx) {

  while(controls_bg != controls_end) {
    mx = mbind(mx, sys(*controls_bg));
    *ret = mx; ++ret; ++controls_bg;
  }
}
```

As we can see the Haskell one-liner translates to a much longer version in C++ and it is also split into two functions. The first one, `macro_trj_init`, initialises the macro trajectory data structure and corresponds to the second argument to `scanr` in the Haskell version. The second one, `macro_trj`, is the implementation of the iterative step (the first argument to `scanr`) in the Haskell version. Here it should be noted that we use the C++ iterator mechanism to implement a function which is even more generic than the equivalent Haskell version: in contrast to the Haskell function, where we handle all sequences as Haskell lists, we do not have a fixed sequence type for the input in the C++ version. Besides the two trajectory computations we have also implemented the actual vulnerability computations. The implementation is straightforward but omitted here due to space constraints — the function signature was presented in Sect. 1 and an example use is presented below, in Sect. 4.

4 The Library as a Domain Specific Language

What makes the library described in the last section a domain specific language embedded in C++? To clarify this we want to emphasise that the concept "domain specific language" is more of a mindset than a formal definition to us. Therefore we show here by example how the library can be used by experts from the vulnerability community which do not have a deep technical knowledge of C++.

The user can use the library by providing objects of appropriate types to plug them in into the library components we provide. When modelling a system the user has to implement a transition function as a monadic coalgebra which is a rather simple task:

```
1  struct InputTransition {
2    typedef std::pair<float,float>    Domain;
3    typedef SimpleProbability<float>  Codomain;
4
5    Codomain operator() (Domain p) {
6      float x = p.second;
7      float delta = p.first;
8      Codomain sp1(x+delta);
9      Codomain sp2(x-delta);
10     Codomain sp3(sp1,sp2,0.75);
11     return sp3;
12   }
13 };
14
15 concept_map MonadicCoalgebraWithInput<InputTransition> { ... }
```

Writing such a function object can be done in an almost algorithmic fashion. First, one has to come up with a name for the function and provide class (in C++, **`struct`** is a class with default public visibility) for it; next, in this class, there have to be two associated types called `Domain` and `Codomain`[4], which define the signature of the function[5]; finally, there has to be an application operator: `Codomain operator() (Domain x) { ... }`. The **`concept_map`** can also be done in a more or less mechanical way. Once such an monadic coalgebra is implemented, it is easy to construct a monadic system out of it. Just create an object of the function type implemented for the monadic coalgebra and use this object in the constructor of an appropriate monadic system type. After being constructed, a monadic system can be used in a vulnerability computation:

[4] One must choose appropriate types for a monadic coalgebra: the domain and codomain have to be related by the `MBindable` concept.

[5] In the current version we only have unary arrows. However, functions which are not unary can be modelled by a unary arrow which has as domain a tuple type.

```
InputTransition inp_trans;
Input_Monadic_System<InputTransition> sys(inp_trans);
Harm h;
Meas m;
vector< vector<float> >  controls;
// fill it, so that we get
// controls = [[3.0], [1.0]]

Meas::Codomain res  =  vulnerability_sys(sys, h, m, 3.0, controls);
```

In this example we assume that we have function objects (implemented in the way described for the transition) `Harm` for the harm computation and `Meas` for the measure function. We do not list them here because they are not interesting.

The library therefore has some typical features of a domain specific language, in particular

- we use a specific terminology;
- the use of the library is quite easy to learn;
- we abstract from technical details, the user does not have to care about the `MBind` operation or the iterators mechanism inside the trajectory computations.

Besides these feature we presented in Sect. 3 that the library comes along with its own type system (a concept hierarchy) and language constructs (generic classes and algorithms). For these reasons we claim that our library implements a domain specific language embedded into C++.

5 Embedding a DSL Using Concepts

In previous sections, we have shown how we developed a generic C++ library for vulnerability assessment based on a high-level, mathematical description of the domain in Haskell (Ionescu, 2009). In this section we outline how a similar process can be applied to other domains, resulting in generic C++ libraries that are efficient but maintain close correspondence with the respective domain. While the process is not fully automated, it is a precise design pattern with parts of it amenable to full automation. Furthermore, parts of the basic framework we have built up for our vulnerability library can be almost directly reused in other developments.

A high-level description, such as the description of vulnerability assessment by Ionescu (2009), consists of several layers:

1. basic mathematical notions on which the description depends;
2. definitions of domain notions;
3. combinators for easy composition of basic building blocks into more complex structures; and
4. domain-specific algorithms that can be applied to the objects in the domain.

In Sect. 3, we describe how each of these layers is designed in our library. In this section, we give a scheme that could be used to derive C++ libraries from Haskell descriptions of other domains. Our examples are from a DSL which is heavily influenced by category theory but the translation scheme could be used also in other cases.

5.1 Basic Types

The mathematical description of the vulnerability assessment domain depends on some basic mathematical notions (see Chapt. 2 Ionescu, 2009, for a complete discussion). Some of these basic notions are directly represented in Haskell, others, such as set membership, for example, are implicit; in particular, the representations of the basic notions of functions, functors, and monads must be translated from the Haskell description to a C++ representation. Haskell provides direct support for functions with built-in function types of the form `X -> Y` where `X` is the domain of the function and `Y` is the codomain. Support for functors is partially provided by *type constructors* which are type functions of kind `* -> *` where `*` is the kind of non-constructor types. The *object mapping* part of the functor can be expressed by a type constructor, like `[]` or `SimpleProb`. The *function mapping* part is given by the method `fmap` of the `Functor` class (see Sect. 1.1). Another basic concept used in the vulnerability assessment domain (and elsewhere) is the `Monad` class that describes a specific kind of functors (again, see Sect. 1.1). Next, we discuss how Haskell types are represented in our C++ encoding and we give a scheme that can be used to represent type constructor classes, such as `Functor` and `Monad`.

In C++ parts of Haskell types are represented by type parameters in templates. The structure of Haskell types is encoded using the three concepts introduced in Sect. 3, `Arrow`, `ConstructedType`, and `SameTypeConstructor`, along with the built-in C++ concept `SameType`. First, function types structure is encoded (Alg. 1.), then, type constructors (Alg. 2.), and finally, type constructors and type constructor identities (Alg. 3.).

Input: The C++ type `T` to be constrained and the Haskell type `X` to be translated.

Output: A list of concept requirements *ARR* encoding the arrow structure of `X`.

Notation: An application of the *Arrow Structure* algorithm is denoted by $as(\texttt{T},\texttt{X})$.

A1. [Generate arrow requirement.] If `X` is a function type of the form `Y -> Z`, generate a requirement `Arrow<T>` and add it to *ARR*.

A2. [Recursive application.] Generate requirements $as(\texttt{Arrow<T>::Codomain},\texttt{Y})$ and $as(\texttt{Arrow<T>::Domain},\texttt{Z})$. Add the generated constraints to *ARR*.

Algorithm 1. Arrow Structure

The above translation of types into concept requirements is then used in translation of all other constructs. Algorithm 4. gives the translation scheme for constructor type classes, used to generate C++ representation of Haskell `Functor` and `Monad`.

5.2 Domain Notions

Translation of domain notions is mostly an application of the algorithms introduced in the previous section. In the vulnerability assessment domain, every domain notion corresponds to a computation and is described using a Haskell function type. For example, a coalgebra is defined as a function of type `Functor f => x -> f x` and a monadic system is defined as a function of type `Monad m => t -> x -> m x`, where

Input: A C++ type `T`, the list of arrow constraints *ARR* generated by Alg. 1. for `T`, and the Haskell type `X`, represented by `T`.

Output: A list of concept requirements *TC* encoding type constructors in `X`.

A1. [Find type constructors.] Scan the type `X` for occurrences of type constructor applications.
A1–1. If the type constructor application is on the left of a function type arrow (`->`), generate a constraint `ConstructedType<A::Domain>`, where `A` is the constraint in *ARR* corresponding to the arrow.

A1–2. If the type constructor application is on the right of a function type arrow (`->`), generate a constraint `ConstructedType<A::Codomain>`, where `A` is the constraint in *ARR* corresponding to the arrow.

A1–3. If type constructor is not a part of a function type (`X` is just a constructor application) then generate a constraint `ConstructedType<T>`.

Algorithm 2. Type Constructors

Input: A C++ type `T`, the list of arrow constraints *ARR* generated by Alg. 1. for `T`, the list of type constructor constraints *TC* generated by Alg. 2. for `T`, and the Haskell type `X` represented by `T`.

Output: A list of concept requirements *S* encoding type and type constructor identities.

A1. [Types.] For every pair of types in `X` named identically, generate a constraint `SameType<A, B>`, where `A` and `B` are drawn from $ARR \cup TC$ in the following fashion:

- If `A` (or `B`) refers to a type occurring in a type constructor, then `A` (or `B`) is `X::Inner`, where `X` is the constraint drawn from *TC* that corresponds to the constructor application.
- If `A` (or `B`) refers to a type on the left of a function type arrow (`->`), then `A` (or `B`) is `X::Domain`, where `X` is the constraint drawn from *ARR* that corresponds to the arrow.
- If `A` (or `B`) refers to a type on the right of a function type arrow (`->`), then `A` (or `B`) is `X::Codomain`, where `X` is the constraint drawn from *ARR* that corresponds to the arrow.

Algorithm 3. Type and constructor identities

some additional conditions have to be satisfied (see Sect. 2.2). Such domain notions are translated to concepts with a single type parameter, with appropriate concept constraints reflecting the Haskell type of a notion. Both the type structure and the context requirements (context is given before `=>` in Haskell types) are translated. The first step, of specifying type structure, requires application of the algorithms from the previous section, resulting in an appropriate list of requirements to be included in a domain notion concept. The second step, in which the context is translated, can be itself split into two steps.

First, non-constructor classes are translated to concept refinement. Note, that a description such as that of Ionescu (2009) makes certain context requirement implicit. For example, the definition of monadic coalgebra given by Ionescu states that a monadic coalgebra is also a coalgebra, but this requirement is not explicitly reflected in the type. In our C++ translation, the monadic coalgebra concept indeed explicitly refines the coalgebra concept: `MonadicCoalgebra<class Arr> : Coalgebra<Arr>`.

Next, after non-constructor context is translated, constructor classes are translated. Since in our representation constructors can be only "seen" in their applications, we cannot directly translate a context such as `Functor f` or `Monad m`. Instead, our translation includes as many operation requirements (e.g., `FMappable` or `MBindable`) as possible in a given context. For example, the `MonadicCoalgebra` concept includes two monad-related requirements (see Sect. 3.1). Consequently, a client of the `MonadicCoalgebra` concept must explicitly state any monad-related requirements that are not already given in `MonadicCoalgebra`.

Input: A constructor type class `Cs` with polymorphic methods.

Output: A set R of C++ concepts, one for each method in `Cs`.

Auxiliary operations:

`gen-name(cname,mname)`, where `cname` is the name of the class `Cs` and `mname` is the name of a method in that class. In our translation scheme `gen-name(Functor,fmap)` generates `FMappable`, for example, but a different naming scheme can be substituted in other situations.

`gen-name(t)`, where `t` is a Haskell type. In our scheme, Haskell types are rewritten in uppercase, type constructors are concatenated with their arguments and rewritten in uppercase, and function types are rewritten to `Arr` and `Arr1`, `Arr2`, and so on, if there is more then one functions that need to be named.

`gen-name(cname)`, where `cname` is the name of a C++ concept. In our translation scheme, `FMappable` generates `FMap`, for example.

A1. [Iterate methods.] For each method `m` in `Cs`:

A1-1. Generate a concept `Ct` named `gen-name(Cs,m)` with a list of parameters `P`, containing `gen-name(`x_i`)` when `m` has a type of the form x_1 `->...->` x_n (note that x_i may be an arrow itself if parentheses occur in the Haskell type) and simply `gen-name(t)` when the type `t` of `m` does not contain arrows.

A1-2. Place constraints in the concept `Ct` by successively applying Alg. 1., Alg. 2., and Alg. 3..

A1-3. Add an associated function `fn` named `gen-name(Ct)` to the concept `Ct`. The return type of `fn` is the last type parameter in `P` and the rest of type parameters are argument types of `fn` (**`void`** if `P` contains only one type argument).

A1-4. Add `Ct` to R.

Algorithm 4. Constructor type classes

Finally, in the case of the `MonadicSystem` concept, we translate monadic system laws into C++ concept axioms. Currently, the translation is not easily mechanisable. We expect that laws given as QuickCheck predicates can be mapped almost directly to concept axioms but this is currently a future work item.

5.3 Combinators

Combinators are implemented as generic classes, the domain concepts specify the type parameters used to construct these new types. The constructed types are implemented

Input: A Haskell function combining two types (which we translated into C++ concepts) into a third one `comb :: X1 -> X2 -> X3`

Output: A generic C++ template class for this function `comb`

A1. [Template specifications] Write a template class with two template parameters and specify their type by an requirements. Furthermore identify all type identities between the template parameters or associated types of template parameters and state them by `SameType` requirements. Put additional necessary requirements.

A2. [Private members] Put two private members of the two template types into the class.

A3. [Input Constructor] Implement a input constructor taking two arguments of the two template parameter types copies them.

A4. [Class structure] Implement the required structure of the class in order to model the result type of the Haskell function. Put associated types and necessary member operations.

Algorithm 5. Translating a Haskell combinator function into a C++ combinator class

as classes which provide the needed infrastructure and operations. In Alg. 5. we give a general translation scheme for combinator operations. In the library this scheme was in particular used to translate different arrow compositions, and domain combinators such as `Monadic_System_Input` (see Sect. 3.2).

The translation scheme application is quite visible in this case. To make explicit that the constructed result type (in this example `Composed_Arrow`) is a model of the required concept, a (straightforward) concept map is needed.

5.4 Algorithms

Algorithms are implemented as generic functions, where we ensure proper use of them by specifying their arguments with concepts for domain notions. For every generic function of the Haskell model we implemented at least one generic C++ function (for technical reasons some Haskell functions are translated into more than one C++ function, see for instance the macro trajectory computations).

In Alg. 6. we present how we can translate the signatures of generic Haskell functions into C++ template specifications. The implementation of such functions remains a creative task to the programmer. However, in some cases, for example for the function `vulnerability_sys`, the translation of the implementation is straightforward.

6 Related Work

In the realm of generic programming, the use of concepts (or type classes) to abstract from representations of data or to capture certain properties of types is a common practise. For example, the C++ Standard Template Library (STL) (Austern, 1998; Stepanov and Lee, 1994) introduces a hierarchy of concepts for data containers and for iterators used to generically traverse ranges of data, and the Boost Graph Library (Siek et al., 2002) introduces concepts representing different kinds of graphs. In the traditional approach, functional notions such as monadic system, for example, are either

Input: A Haskell function with signature `f :: X1 -> X2 -> ... -> Xn`
Output: A generic C++ template specification for this function `f`

A1. [Template type parameters] Introduce a template parameter for each variable in the type of `f`.

A2. [Template function parameters] Identify all X_i in `f` which are functions and put a template parameter for them.

A3. [Template function parameters requirements] For every type parameter which encodes a function type, put a requirement stating its type (`Arrow` or `MonadicCoalgebra` and so on).

A4. [Type constructor requirements] Identify all template parameters or associated types of template parameters which are constructed types and put a `ConstructedType` requirement on them.

A5. [Type equality] Identify all type identities between template parameters or associated types of template parameters which have to be stated explicitly and put a `SameType` requirement for every necessary identity.

A6. [Type constructor equality] Identify all type constructor identities a `SameTypeConstructor` requirement for them.

A7. [Type relations] Identify all pairs of (monadic) Coalgebras and template parameters or associated types of template parameters which have to be used in a fmap or a bind operation and put a `FMappable` or a `MBindable` requirement for them.

Algorithm 6. Generic C++ function signature from a Haskell function

not represented explicitly at all (in C++ libraries) or are represented by a particular type (as in Haskell libraries). In our generic library for vulnerability assessment, we take abstraction further in conceptually representing functions as well as data. In Haskell, almost all libraries are generic in the sense that they use and define type classes. Some more advanced examples are (Chakravarty et al., 2005; Claessen and Hughes, 2000; Jansson and Jeuring, 2002; Oliveira et al., 2006)

Recently, new concept applications have been enabled (and inspired) by the linguistic support for concepts in C++0x (Gregor et al., 2006). For example, much of the standard library of C++ has been conceptualised, that is, rewritten using the new concepts feature (see for example Gregor et al. (2008a,b)). This body of code and specifications provides a good example of how generic libraries will be written when concepts officially become a feature of C++. Our work goes further than STL concepts. The first difference is that we systematically tackle type constructor concepts; the only similar concept in STL is the `Allocator` concept (Halpern, 2008), which requires compiler support to verify constraints that are expressed using `SameTypeConstructor` concept in our framework. The other difference is in handling of computations: in our framework a computation is represented by a self-contained `Arrow` concept that knows its domain and codomain types, while in the conceptualised STL a computation is represented by a `Callable` family of concepts that require domain types as concept arguments.

One aspect of our work is implementation of functional programming constructs in C++. Much similar work has been done before, including the Boost Lambda library (Järvi et al., 2003), FC++ library (McNamara and Smaragdakis, 2004), and Boost Fusion library (de Guzman and Marsden, 2009). All the previous approaches rely heavily on template meta-programming, while we encode the types of functions on the level of concepts.

Hudak (1996) proposes the use of domain-specific embedded languages. In this approach, the syntactic mechanisms of the base language are used to express the idioms of a domain; examples of such languages include FPIC (Kamin and Hyatt, 1997), a language for picture drawing, and a robot control language (Peterson et al., 1998). Our library can be considered an embedded language as well but instead of exploiting syntactic mechanisms of the language we rely on the concept-checking mechanisms to facilitate domain idioms.

This paper can also be seen as continuing the line of work relating concepts in Haskell and C++: Bernardy et al. (2008); Garcia et al. (2007); Zalewski et al. (2007).

7 Design Questions and Future Work

Our C++ representation of domain notions is based on some specific design choices. In this section, we discuss our representation of functions and domain-level algorithms, as two examples of such choices. For each, we discuss advantages and disadvantages of competing designs. Such consideration is one of the main directions of our future research. Making design choices is not limited to C++; in the last part of this section, we briefly outline how a generic library similar to the one described in this paper could be developed in Haskell.

7.1 Functions

Functions are represented by the `Arrow` concept. In this case there is the alternative to represent functions by a concrete datatype, for instance the function type provide by the Boost library[6]. The use of such a function type could make the code at some points more readable and understandable, however we have decided to use no function types for two reasons: first of all, every function type in C++ introduces inefficiency due to the internal use of function pointers. And secondly the use of a specific type would decrease the level of genericity: everyone who wants to use our library would be bound to this specific type. Using a concept instead of a type gives more flexibility. If a user wants to use a specific type for functions he just has to declare this type to be an instance of our arrow concept. Concept maps for common function types could be part of the library in future versions.

Otherwise the design of the concept `Arrow` is not unique. In our version the type of the domain and of the codomain are associated types. One could also make the domain type (or both types) a parameter of the concept. The `std::CallableX` concepts do this:

```
1 concept Callable1<typename F, typename T1> {
2   typename result_type;
3   result_type operator()(F&, T1);
4 };
```

We have chosen our version because it seems to be closer to mathematical ideas. However there are advantages of the `CallableX` family of concepts, for instance they

[6] `http://www.boost.org/`

allow different calling conventions for the same function object. Exploring all the implications of the different design of these two concepts is beyond the scope of this paper and will be subject to future work.

7.2 Representation of Domain-Level Algorithms

Domain-level algorithms can be encoded as concepts with associated functions or concepts with associated `Arrow` types. For instance, the bind operation of monads, `MBindable`, can be implemented as show in Sect. 3, or with an associated type representing a function of the type `MBind_Arr :: m x -> m y`:

```
concept MBindable<class MX, class Arr> {
  requires ConstructedType<MX>, Arrow<Arr>,
           ConstructedType<Arr::Codomain>,
           SameType<MX::Inner,Arr::Domain>,
           SameTypeConstructor<MX,Arr::Codomain>;

  Arrow MBind_Arr;
  requires SameType<MX,MBind_Arr::Domain>,
           SameType<Arr::Codomain,MBind_Arr::Codomain>;
};
```

An associated function can be directly used in a generic algorithm, while the associated function type requires a constructed object before it can be used. On the other hand, an associated function cannot be easily used in higher-order combinators, where a function object can be passed as a parameter.

7.3 From Haskell to C++ and Back Again

Recent additions of support for associated types to the GHC Haskell compiler (Chakravarty et al., 2005; Schrijvers et al., 2008) allows us to express the C++ concept framework also in Haskell (with extensions). We have ported parts of our C++ implementation "back to Haskell" and interesting future work would be to complete this port and compare the expressivity and efficiency of the generated code. This is a sample of the Haskell version using associated types.

```
class Arrow arr where
  type Domain arr
  type Codomain arr
  (!) :: arr -> (Domain arr -> Codomain arr)
```

References

C++ Standard, ISO/IEC 14882:2003(E). ANSI-ISO-IEC, ANSI standards for information technology edition (2003)

Austern, M.H.: Generic Programming and the STL: Using and Extending the C++ Standard Template Library. Addison-Wesley, Reading (1998)

Bernardy, J.-P., Jansson, P., Zalewski, M., Schupp, S., Priesnitz, A.: A comparison of C++ concepts and Haskell type classes. In: Proc. ACM SIGPLAN Workshop on Generic Programming, pp. 37–48. ACM, New York (2008)

Brooks, N.: Vulnerability, risk and adaptation: A conceptual framework. Tyndall Center Working Paper 38 (2003)

Chakravarty, M.M.T., Keller, G., Jones, S.P.: Associated type synonyms. In: ICFP 2005: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming, pp. 241–253. ACM, New York (2005)

Claessen, K., Hughes, J.: QuickCheck: A lightweight tool for random testing of Haskell programs. In: ICFP 2000: International Conference on Functional Programming, pp. 268–279. ACM, New York (2000)

Garcia, R., Jarvi, J., Lumsdaine, A., Siek, J., Willcock, J.: An extended comparative study of language support for generic programming. J. Funct. Program 17(2), 145–205 (2007)

Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Dos Reis, G., Lumsdaine, A.: Concepts: Linguistic support for generic programming in C++. In: Proc. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 291–310. ACM Press, New York (2006)

Gregor, D., Marcus, M., Witt, T., Lumsdaine, A.: Foundational concepts for the C++0x standard library (revision 4). Technical Report N2737=08-0247, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++ (August 2008a)

Gregor, D., Siek, J., Lumsdaine, A.: Iterator concepts for the C++0x standard library (revision 4). Technical Report N2739=08-0249, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++ (August 2008b)

Gregor, D., Stroustrup, B., Widman, J., Siek, J.: Proposed wording for concepts (revision 8). Technical Report N2741=08-0251, ISO/IEC JTC1/SC22/WG21 - C++ (August 2008c)

de Guzman, J., Marsden, D.: Fusion library homepage (March 2009), `http://www.boost.org/libs/fusion`

Halpern, P.: Defects and proposed resolutions for allocator concepts. Technical Report N2810=08-0320, ISO/IEC JTC1/SC22/WG21 - C++ (December 2008)

Hudak, P.: Building domain-specific embedded languages. ACM Comput. Surv., 196 (1996)

Ionescu, C.: Vulnerability Modelling and Monadic Dynamical Systems. PhD thesis, Freie Universität Berlin (2009)

Janssen, M.A., Ostrom, E.: Resilience, vulnerability and adaptation: A cross-cutting theme of the international human dimensions programme on global environmental change. Global Environmental Change 16(3), 237–239 (2006) (Editorial)

Jansson, P., Jeuring, J.: Polytypic data conversion programs. Science of Computer Programming 43(1), 35–75 (2002)

Järvi, J., Powell, G., Lumsdaine, A.: The lambda library: Unnamed functions in C++. Software: Practice and Experience 33(3), 259–291 (2003)

Kamin, S.N., Hyatt, D.: A special-purpose language for picture-drawing. In: Proc. Conference on Domain-Specific Languages (DSL), pp. 297–310. USENIX Association (1997)

McNamara, B., Smaragdakis, Y.: Functional programming with the FC++ library. J. of Functional Programming 14(4), 429–472 (2004)

Oliveira, B.C.d.S., Hinze, R., Löh, A.: Extensible and modular generics for the masses. In: Nilsson, H. (ed.) Trends in Functional Programming, pp. 199–216 (2006)

Peterson, J., Hudak, P., Elliott, C.: Lambda in motion: Controlling robots with haskell. In: Gupta, G. (ed.) PADL 1999. LNCS, vol. 1551, pp. 91–105. Springer, Heidelberg (1999)

Petoukhov, V., Ganopolski, A., Brovkin, V., Claussen, M., Eliseev, A., Kubatzki, C., Rahmstorf, S.: CLIMBER-2: a climate system model of intermediate complexity. Part I: model description and performance for present climate. Climate dynamics 16, 1–17 (2000)

Peyton Jones, S. (ed.): Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press, Cambridge (2003)

Schrijvers, T., Peyton Jones, S., Chakravarty, M., Sulzmann, M.: Type checking with open type functions. In: ICFP 2008: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, pp. 51–62. ACM, New York (2008)

Siek, J.G., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley, Reading (2002)

Stepanov, A.A., Lee, M.: The Standard Template Library. Technical Report HPL-94-34, Hewlett-Packard Laboratories, Revised in, as tech. rep. HPL-95-11 (May 1994)

Stroustrup, B.: The C++ Programming Language, 3rd edn. Addison-Wesley, Reading (1997)

Vandervoorde, D., Josuttis, N.M.: C++ Templates: The Complete Guide. Addison-Wesley, Amsterdam (2002)

Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proc. 16th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL), pp. 60–76. ACM Press, New York (1989)

Wolf, S., Lincke, D., Hinkel, J., Ionescu, C., Bisaro, S.: A formal framework of vulnerability. Final deliverable to the ADAM project. FAVAIA working paper 8, Potsdam Institute for Climate Impact Research, Potsdam, Germany (2008),
`http://www.pik-potsdam.de/favaia/pubs/favaiaworkingpaper8.pdf`

Zalewski, M., Priesnitz, A.P., Ionescu, C., Botta, N., Schupp, S.: Multi-language library development: From Haskell type classes to C++ concepts. In: Striegnitz, J. (ed.) Proc. 6th Int. Workshop on Multiparadigm Programming With Object-Oriented Languages (MPOOL) (July 2007)