Playing with type classes

Based on the paper "Generic Libraries in C++ with Concepts from High-Level Domain Descriptions in Haskell — A Domain-Specific Library for Computational Vulnerability Assessment"

Patrik Jansson, fp.set.cse.chalmers.se

Chalmers FP workshop 2009

(Joint work with M. Zalewski (was at Chalmers), D. Lincke and C. Ionescu at PIK = Potsdam Institute for Climate Impact Research.)

Application area: Computational Vulnerability Assessment

- Complex models: ocean, atmosphere, biosphere, economy etc.
- Provide basic data for political decisions in the climate area.
- Measure the possible harm of future evolutions

Application area: Computational Vulnerability Assessment

- Complex models: ocean, atmosphere, biosphere, economy etc.
- Provide basic data for political decisions in the climate area.
- Measure the possible harm of future evolutions

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

Measure possible harm

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

Measure possible harm

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

```
type Evolution = [State]
possible :: State -> M Evolution
harm     ::            Evolution -> Harm
measure  ::                       M Harm -> V
```

Measure possible harm

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

```
type Evolution = [State]
possible :: State -> M Evolution
harm     ::            Evolution -> Harm
measure  ::                       M Harm -> V
```

Examples:

```
type M = Id          -- Deterministic system
type M = [ ]         -- Non-deterministic system
type M = SimpleProb  -- Probabilistic system
newtype SimpleProb a = SP [(a, Double)]
```

Measure possible harm

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

```
type Evolution = [State]
possible :: State -> M Evolution
harm     ::            Evolution -> Harm
measure  ::                       M Harm -> V
```

Examples:

```
type M = Id          -- Deterministic system
type M = [ ]         -- Non-deterministic system
type M = SimpleProb  -- Probabilistic system
newtype SimpleProb a = SP [(a, Double)]
```

```
type Harm = (LivesLost, EconomicLoss)
type V    = Harm
```

Calculate possible evolutions

```
possible :: State -> M [State]
```

```
possible = micro_trj sys inputs
micro_trj :: (Monad m) => (i -> (x -> m  x)) ->
                          [i] -> x -> m [x]
```

Calculate possible evolutions

```
possible :: State -> M [State]
```

```
possible = micro_trj sys inputs
micro_trj :: (Monad m) => (i -> (x -> m  x)) ->
                          [i] -> x -> m [x]
```

We model a *monadic dynamic system* (MDS) as a function

```
sys :: (Monad m) =>  i -> (x -> m x)
```

where `i` is for input (for example greenhouse gas emission / year)

From Haskell to C++

```
sys :: (Monad m) =>  i -> (x -> m x)
```

This Haskell model uses

- a constructor class `Monad m`,
- a type constructor application `m x`,
- a monadic transition function `x -> m x` and
- currying `i -> (x -> m x)`.

We represent all of these in C++ + concepts (with some effort...).

Functions / arrows in C++ and Haskell

We model a type like `a -> b` with some `Arr` in the concept `Arrow1`.

```
concept Arrow1<class Arr> { // in C++0x
  typename Domain;
  typename Codomain;
  Codomain operator () (Arr, Domain);
};
```

Functions / arrows in C++ and Haskell

We model a type like `a -> b` with some `Arr` in the concept `Arrow1`.

```
concept Arrow1<class Arr> { // in C++0x
  typename Domain;
  typename Codomain;
  Codomain operator () (Arr, Domain);
};
```

or with a type class

```
class Arrow1 arr where        -- in Haskell
  type Domain arr
  type Codomain arr
  (!) :: arr -> (Domain arr -> Codomain arr)
```

I will mainly use Haskell syntax but the library is written in C++.

Arrow instances

Exercise: Normal functions are arrows . . .

Arrow instances

Exercise: Normal functions are arrows ...

```
instance Arrow1  (a->b) where
  type Domain   (a->b) = a
  type Codomain (a->b) = b
  f ! x = f x
```

Arrow instances

Exercise: Normal functions are arrows . . .

```
instance Arrow1   (a->b) where
  type Domain    (a->b) = a
  type Codomain (a->b) = b
  f ! x = f x
```

. . . Exercise: pairs can also be seen as arrows (from `Bool`)

Arrow instances

Exercise: Normal functions are arrows . . .

```
instance Arrow1  (a->b) where
  type Domain   (a->b) = a
  type Codomain (a->b) = b
  f ! x = f x
```

. . . Exercise: pairs can also be seen as arrows (from `Bool`)

```
instance Arrow1  (a,a) where
  type Domain   (a,a) = Bool
  type Codomain (a,a) = a
  (f,t) ! b = if b then t else f
```

Arrow instances

Exercise: Normal functions are arrows ...

```
instance Arrow1  (a->b) where
  type Domain   (a->b) = a
  type Codomain (a->b) = b
  f ! x = f x
```

...Exercise: pairs can also be seen as arrows (from `Bool`)

```
instance Arrow1  (a,a) where
  type Domain   (a,a) = Bool
  type Codomain (a,a) = a
  (f,t) ! b = if b then t else f
```

...and finite maps are arrows

```
import qualified Data.Map as FM
instance Ord a => Arrow1  (FM.Map a b) where
  type Domain   (FM.Map a b) = a
  type Codomain (FM.Map a b) = b
  (!) = (FM.!)
```

Other encodings of functions

We have three types (`t`, `dom`, `cod`) where `t = dom -> cod`.

Other encodings of functions

We have three types (t, dom, cod) where t = dom -> cod.
Each t uniquely determines dom and cod

```
class Arrow1 t where type Dom t; type Cod t
  (!) :: arr -> (Domain arr -> Codomain arr)
```

Exercise: write the same with functional dependencies, without associated types.

Other encodings of functions

We have three types (`t`, `dom`, `cod`) where `t = dom -> cod`.
Each `t` uniquely determines `dom` and `cod`

```
class Arrow1 t where type Dom t; type Cod t
  (!) :: arr -> (Domain arr -> Codomain arr)
```

Exercise: write the same with functional dependencies, without associated types.

```
class Arrow1 t dom cod | t -> dom cod where
  (!) :: t -> dom -> cod
```

Other encodings of functions, contd.

Exercise: Each pair `t`, `dom` determines `cod`

Other encodings of functions, contd.

Exercise: Each pair `t`, `dom` determines `cod`

```
class Arrow2 t dom where          type Cod t dom
```

Other encodings of functions, contd.

Exercise: Each pair `t`, `dom` determines `cod`

```
class Arrow2 t dom where          type Cod t dom
```

Exercise: All combinations of `t`, `dom`, `cod` possible

Other encodings of functions, contd.

Exercise: Each pair `t`, `dom` determines `cod`

```
class Arrow2 t dom where          type Cod t dom
```

Exercise: All combinations of `t`, `dom`, `cod` possible

```
class Arrow3 t dom cod where
```

Other encodings of functions, contd.

Exercise: Each pair t, dom determines cod

```
class Arrow2 t dom where          type Cod t dom
```

Exercise: All combinations of t, dom, cod possible

```
class Arrow3 t dom cod where
```

Exercise: Each q determines the mapping from dom, cod to t

Other encodings of functions, contd.

Exercise: Each pair `t`, `dom` determines `cod`

```
class Arrow2 t dom where          type Cod t dom
```

Exercise: All combinations of `t`, `dom`, `cod` possible

```
class Arrow3 t dom cod where
```

Exercise: Each `q` determines the mapping from `dom`, `cod` to `t`

```
class Arrow (q::*->*->*) where -- t = q a b
```

Other encodings of functions, contd.

Exercise: Each pair `t`, `dom` determines `cod`

```
class Arrow2 t dom where          type Cod t dom
```

Exercise: All combinations of `t`, `dom`, `cod` possible

```
class Arrow3 t dom cod where
```

Exercise: Each `q` determines the mapping from `dom`, `cod` to `t`

```
class Arrow (q::*->*->*) where -- t = q a b
```

Exercise: Each `dom` determines the mapping from `cod` to `t`

Other encodings of functions, contd.

Exercise: Each pair t, dom determines cod

```
class Arrow2 t dom where          type Cod t dom
```

Exercise: All combinations of t, dom, cod possible

```
class Arrow3 t dom cod where
```

Exercise: Each q determines the mapping from dom, cod to t

```
class Arrow (q::*->*->*) where -- t = q a b
```

Exercise: Each dom determines the mapping from cod to t

```
class ArrowFrom dom where type Tab dom :: * -> *
```

Examples of ArrowFrom instances

```
class ArrowFrom dom where
  type Tab dom :: * -> *
  (!) :: (Tab dom cod) -> (dom -> cod)
```

Examples of ArrowFrom instances

```
class ArrowFrom dom where
  type (:->) dom :: * -> *
  (!) :: (dom :-> cod) -> (dom -> cod)
```

Exercise: `ArrowFrom Bool`

Examples of ArrowFrom instances

```
class ArrowFrom dom where
  type (:->) dom :: * -> *
  (!) :: (dom :-> cod) -> (dom -> cod)
```

Exercise: `ArrowFrom Bool`

```
data Two a = Two a a
instance ArrowFrom Bool where
  type (:->) Bool = Two
  Two f t ! False = f
  Two f t ! True  = t
```

Examples of ArrowFrom instances

```
class ArrowFrom dom where
  type (:->) dom :: * -> *
  (!) :: (dom :-> cod) -> (dom -> cod)
```

Exercise: `ArrowFrom Bool`

```
data Two a = Two a a
instance ArrowFrom Bool where
  type (:->) Bool = Two
  Two f t ! False = f
  Two f t ! True  = t
```

Exercise: `ArrowFrom Nat`

Examples of ArrowFrom instances

```
class ArrowFrom dom where
  type (:->) dom :: * -> *
  (!) :: (dom :-> cod) -> (dom -> cod)
```

Exercise: `ArrowFrom Bool`

```
data Two a = Two a a
instance ArrowFrom Bool where
  type (:->) Bool = Two
  Two f t ! False = f
  Two f t ! True  = t
```

Exercise: `ArrowFrom Nat`

```
data Nat      = Z | S Nat
data Stream a = a :< Stream a
ones = 1 :< ones
instance ArrowFrom Nat where
  type (:->) Nat = Stream
  a :< _   ! Z     =  a
  _ :< as  ! S n   =  as ! n
```

More ArrowFrom instances

```
data Both f g c = Both (f c) (g c)
instance (ArrowFrom a, ArrowFrom b) =>
          ArrowFrom (Either a b) where
  type (:->) (Either a b) = Both ((:->) a)
                                 ((:->) b)
  Both l r  !  e  = either (l!) (r!) e
```

More ArrowFrom instances

```
data Both f g c = Both (f c) (g c)
instance (ArrowFrom a, ArrowFrom b) =>
           ArrowFrom (Either a b) where
  type (:->) (Either a b) = Both ((:->) a)
                                 ((:->) b)
  Both l r  !  e  = either (l!) (r!) e
```

```
newtype Compose f g c = Compose (f (g c))
instance (ArrowFrom a, ArrowFrom b) =>
           ArrowFrom (a, b) where
  type (:->) (a, b) = Compose ((:->) a)
                              ((:->) b)
  Compose x ! (a, b) = x ! a ! b
```

Other concepts / type classes

```
class    (Arrow carr, Arrow (Codomain carr)) =>
         CurriedArrow carr
instance (Arrow carr, Arrow (Codomain carr)) =>
         CurriedArrow carr
```

```
class ConstructedType t where
    type Inner t

instance Functor f => ConstructedType (f a)
  where type Inner (f a) = a
```

Monads get ugly

```
class ( Arrow arr
      , ConstructedType mx
      , ConstructedType (Codomain arr)
      , SameType (Inner mx) (Domain arr)
      , SameTypeConstructor mx (Codomain arr)
      ) => MBindable mx arr where
  mbind :: mx -> arr -> Codomain arr

-- sanity check
instance (Functor m, Monad m) =>
         MBindable (m a) (a -> m b) where
  mbind = (>>=)

class ConstructedType mx => MReturnable mx where
  mreturn :: Inner mx -> mx
```

Contributions

- a simple model for vulnerability assessment
- concepts (type classes) for functions, functors, monads, etc.
- deeper understanding of generic programming by contrasting Haskell and C++

Algebra of monadic dynamical systems

(or an Algebra of indexed co-algebras)

Given `sx :: x -> m x` and `sy :: y -> m y` we define

```
lockstep sx sy :: (x,y)-> m (x,y)
```

(forward) Kleisli composition

```
(>=>) :: (x -> m y)-> (y -> m z)-> (x -> m z)
```

Communicating systems

```
compose_sys :: Monad m =>
               ((t,t1) -> (x1 -> m x1)) ->
               ((t,t2) -> (x2 -> m x2)) ->
               (x1 -> t2) -> (x2 -> t1) ->
               t -> (x1,x2) -> m (x1, x2)
compose_sys sys1 sys2 p1 p2 t (x1, x2) =
  liftM2 (,) (sys1 (t, p2 x2) x1)
             (sys2 (t, p1 x1) x2)
```

The two systems `sys1` and `sys2` share a dependency on `t`. They both have their own "local" input `ti` and state `xi`. The two projection functions `p1` and `p2` implement a coupling between the two systems. In the combined systems the "local" inputs are hidden and the only remaining input is the global input `t`.