Advanced Programming (I00032)
Extendable DSLs and GADT

Assignment 12

Extendable DSLs

In the lecture we showed various ways to construct expandable DSLs. Our approach to a shallow extendable DSL was not that successful. Using classes it is possible to build the DSL incrementally, but the specific form of expressions, as functions of type `t → t`, imposed undesirable restrictions on the use of various types. In this assignment you implement a different approach that does not suffer from this problem.

The DSL is composed of the following parts:

Arithmetic expressions These are expressions of any type. The expression is either a constant, `Lit`, an addition, `+.`, or a multiplication `*.`. Ensure that this works at least for integers and Booleans. The `+` for Booleans is the logical OR, `||`, and the `*` is the logical AND, `&&`.

Store handling A `read` and `write` of integer values to the single storage cell. We can of course provide and expression of type `Int` to `write`, this expression will be evaluated before the value can be written to the store.

Truth Values Apart from the overloaded OR and AND operator there is negation, `-.`, and the `XOR` of Boolean values. The result of `x XOR y` is `True` when exactly one of its arguments in `True`.

Equality We can compare the value of arbitrary expressions with the operator `=.=`.

Exceptions We can `throw` an (unparameterized) exception. When it is thrown in the first argument of a `try` the second argument of this `try` will be executed. Without an exception this second argument will be ignored during evaluation.

Arithmetic expressions, `aexpr`, can consists of all classes of operations except `truth`. Boolean expressions, `bexpr`, can consists of all classes of operations except `store`. The necessary class definitions of our DSL are:

```
class arith x where
    lit :: a → x a | toString a
    (+.) infixl 6 :: (x a) (x a) → x a | + a // integer addition, Boolean OR
    (*.) infixl 7 :: (x a) (x a) → x a | * a // integer multiplication, Boolean AND
class store x where
    read  :: (x Int)
    write :: (x Int) → x Int
class truth x where
    (XOR) infixr 3 :: (x Bool) (x Bool) → x Bool
    -.    :: (x Bool) → x Bool
class (=.=) infix 4 x :: (x a) (x a) → x Bool |==a
class except x where
```

```
    throw :: (x a)
    try   :: (x a) (x a) → x a

class aexpr x | arith, store, except, =.= x
class bexpr x | arith, truth, except, =.= x
class expr x  | aexpr, bexpr x
```

1 Showing expressions

It should be straightforward to show arbitrary expressions by an instance of these classes for `:: Show a = Show ([String] → [String])`. Apart from the additional constructor `Show` this is analogous to the example shown in the lecture.

2 Evaluation

In the evaluation we deviate slightly from the approach in the lecture. To prevent problems with expressions of different types we provide only the `State` as an argument to the step-function. The `Step` is defined as:

```
:: Step a = Step (State → (Maybe a, State))
:: State :== Int
```

It is convenient to implement an instance of the class `Monad` for `Step`.

Based on this monad the implementation of the instance for `Step` of our classes should follow the familiar pattern.

3 Examples

Based on these definitions we can write expressions like:

```
seven :: e Int | aexpr e
seven = lit 3 +. lit 4

throw1 :: e Int | expr e
throw1 = lit 3 +. throw

six :: e Int | expr e
six = write (lit 3) +. read

try1 :: e Int | expr e
try1 = try throw1 (lit 42)

loge :: e Bool | expr e
loge = lit True *. -. (lit True)

comp :: e Bool | expr e
comp = lit 1 =.= lit 2 XOR -. (-. (lit True))
```

The good thing is that the compiler is able to derive suitable types for these expressions. Verify that showing and evaluation of these expressions yield the correct results. Include these results as a comment in your program.

Verify that ill-typed expressions in the DSL are rejected by the Clean compiler.

4 GADTs

Implement the language using the poor-mans GADTs. Building the expressions in a modular way is pretty challenging on its own. Hence, we use a single data type to represent expressions similar to the expressions made by the class `expr` above.

Without the GADT-magic the data type is:

```
:: Expr a
   = Lit a
   | (+.) infixl 6 (Expr a) (Expr a)
   | (*.) infixl 7 (Expr a) (Expr a)
   | Read
   | Write (Expr Int)
   | XOR (Expr Bool) (Expr Bool)
   | Not (Expr Bool)
   | ∃b: Eq (Expr b) (Expr b)
   | Throw
   | Try (Expr a) (Expr a)
```

1. Add the bimap's and class constraints necessary for the GADT manipulations.
2. Make an implementation of **class** `show a :: a [String] → [String]` for `Expr a`.
3. Define a function `eval :: (Expr a) State → (Maybe a, State)` to evaluate these expressions.

Deadline

The deadline for this assignment is December 14, 13:30h.