Techniques for Implementation of Symbolically Interpretable Haskell EDSLs

---

Grigoriy Volkov

May 27th, 2020

ISP RAS / NRU HSE

Introduction

- Haskell is a typed functional programming language famous for hosting embedded domain-specific languages (EDSLs);

Introduction

- Haskell is a typed functional programming language famous for hosting embedded domain-specific languages (EDSLs);
- At ISP RAS, we are developing formal specification languages embedded in Haskell, with some unique requirements;

Introduction

- Haskell is a typed functional programming language famous for hosting embedded domain-specific languages (EDSLs);
- At ISP RAS, we are developing formal specification languages embedded in Haskell, with some unique requirements;
- Notably, we need to be able to interpret programs in these languages symbolically.

Introduction to Haskell

Haskell is a functional language

In functional programming, functions are first-class values that can be passed around to other functions.

```haskell
map (+ 1) [1, 2, 3] -- [2,3,4]
```

Haskell is a functional language

In functional programming, functions are first-class values that can be passed around to other functions.

```
map (+ 1) [1, 2, 3] -- [2,3,4]
```

Composing functions is key to effective functional programming.

```
sumOfSquares = foldr (+) 0 . map (\x -> x * x)

sumOfSquares [1, 2, 3] -- 14
```

Haskell is a functional language

In functional programming, functions are first-class values that can be passed around to other functions.

```
map (+ 1) [1, 2, 3] -- [2,3,4]
```

Composing functions is key to effective functional programming.

```
sumOfSquares = foldr (+) 0 . map (\x -> x * x)

sumOfSquares [1, 2, 3] -- 14
```

These days, even the most popular industrial languages support some functional programming (C# LINQ, Java Streams)!

Haskell is a pure functional language

Haskell functions do not have side effects – they're like mathematical functions, not imperative procedures.

```
incr a = launchRockets && a + 1 -- impossible!
```

Haskell is a pure functional language

Haskell functions do not have side effects – they're like mathematical functions, not imperative procedures.

```
incr a = launchRockets && a + 1 -- impossible!
```

How can we communicate with the real world?

We can **compose** effectful **computations**!

```
main = putStrLn "hello" >> putStrLn "world"
```

Haskell offers powerful abstractions for doing that. (Coming up in a few slides!)

Haskell is a typed functional language

Structures with fields are defined like this:

```
data Book = Book { catalogNumber :: Int, title :: String }
```

These are called product types.

Haskell is a typed functional language

Structures with fields are defined like this:

```
data Book = Book { catalogNumber :: Int, title :: String }
```

These are called product types.

Record syntax (above) is optional, it just defines accessor functions.

```
data Book = Book Int String
```

Haskell is a typed functional language

Structures with fields are defined like this:

```
data Book = Book { catalogNumber :: Int, title :: String }
```

These are called product types.

Record syntax (above) is optional, it just defines accessor functions.

```
data Book = Book Int String
```

Enumerations are called sum types.

```
data Color = Red | Green | Blue
```

Haskell is a typed functional language

Structures with fields are defined like this:

```
data Book = Book { catalogNumber :: Int, title :: String }
```

These are called product types.

Record syntax (above) is optional, it just defines accessor functions.

```
data Book = Book Int String
```

Enumerations are called sum types.

```
data Color = Red | Green | Blue
```

The `data` syntax lets you define **sums of products**: each **constructor** can have any number fields:

```
data Part = CPU { cpuSpeed :: Int, cpuManufacturer :: String }
          | RAM { ramSize :: Int, ramSticks :: Int }
          | Fan
```

This is called **algebraic data types**.

Haskell is a polymorphic functional language

Data types can be parameterized by types:

```
data Maybe a = Just a | Nothing
```

Haskell is a polymorphic functional language

Data types can be parameterized by types:

```
data Maybe a = Just a | Nothing
```

Functions can be parameterized as well.

```
map :: (a -> b) -> [a] -> [b]
```

Haskell is a polymorphic functional language

Data types can be parameterized by types:

```
data Maybe a = Just a | Nothing
```

Functions can be parameterized as well.

```
map :: (a -> b) -> [a] -> [b]
```

This is called **parametric polymorphism** (known as "generics" in Java, "templates" in C++). In a pure language like Haskell, the type signature can make it really obvious what the function would be. (In fact, it is *impossible* for a function with this type signature to be anything valid other than `map`!)

Haskell is a polymorphic functional language

Data types can be parameterized by types:

```
data Maybe a = Just a | Nothing
```

Functions can be parameterized as well.

```
map :: (a -> b) -> [a] -> [b]
```

This is called **parametric polymorphism** (known as "generics" in Java, "templates" in C++). In a pure language like Haskell, the type signature can make it really obvious what the function would be. (In fact, it is *impossible* for a function with this type signature to be anything valid other than `map`!)

There is also **ad-hoc polymorphism** via **typeclasses** (roughly similar to "interfaces" in Java/etc., but more powerful):

```
class Plus a where add :: a -> a -> a

instance Plus Int where add a b = a + b
instance Plus String where add a b = a <> b
```

Fundamentals of Embedded DSLs

Naive EDSL implementation

Declare an Abstract Syntax Tree type, write interpreters that match on it.

```
data Expr = Num Int
          | Add Expr Expr
          | ...
-- e.g. Add (Add (Num 1) (Num 2)) (Num 3)

eval (Num i) = i
eval (Add l r) = eval l + eval r

print (Num i) = show i
print (Add l r) = print l <> " + " <> print r
```

Expression Problem[1]: cannot add new constructs, only new interpretations!

---

[1] The expression problem. / P. Wadler [et al.] // Posted on the Java Genericity mailing list. 1998.

Tagless final encoding[2]

Encode syntax as typeclasses, semantics as instances:

```
class Arithmetic repr where
    num :: Int → repr Int
    add :: repr Int → repr Int → repr Int
-- e.g. add (add (num 1) (num 2)) (num 3)

newtype Eval a = Eval { unEval :: a }
instance Arithmetic Eval where
    num = Run
    add l r = Run $ unRun l + unRun r

newtype Print a = Print { unPrint :: Int → String }
instance Arithmetic Print where
    num = Print . const . show
    add l r = Print $ \c → l c <> " + " <> r c
```

---

[2] *Carette J., Kiselyov O., Shan C.-c.* Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. // J. Funct. Program. USA, 2009. Sept. Vol. 19, no. 5. P. 509–543.

Tagless final encoding, pt. 2

We can add new language constructs as a separate class!

```
class Lambda repr where
    lam :: (repr a → repr b) → repr (a → b)
    app :: repr (a → b) → repr a → repr b

instance Lambda Eval where
    lam f = Run $ unRun . f . Run
    app l r = Run $ (unRun l) (unRun r)

instance Lambda Print where
    lam f = Print $ \c → "\var" <> c <> " → " <>
        (unPrint $ f $ Print $ const $ "var" <> show c) (c + 1)
    app l r = Print $ \c → "(" <> l c <> ") (" <> r c <> ")"
```

Tagless final encoding, pt. 3

Programs are polymorphic in the interpreter type.

We can compose languages by requiring multiple typeclasses to be implemented on the interpreter type!

```
prog :: (Lambda repr, Arithmetic repr) => repr Int
prog = app (lam (add (num 1))) (num 2)
```

Implementing Symbolically Interpretable Imperative EDSLs

The Obligatory Monad Tutorial Slide

You can write imperative programs in Haskell with **do notation**:

```
main = do
    name ← getLine
    putStrLn $ "Hello, " <> name
```

The Obligatory Monad Tutorial Slide

You can write imperative programs in Haskell with **do notation**:

```
main = do
    name ← getLine
    putStrLn $ "Hello, " <> name
```

This is just syntactic sugar for:

```
main = getLine >>= \name → putStrLn $ "Hello, " <> name
```

What is >>=?

The Obligatory Monad Tutorial Slide

You can write imperative programs in Haskell with **do notation**:

```
main = do
    name ← getLine
    putStrLn $ "Hello, " <> name
```

This is just syntactic sugar for:

```
main = getLine >>= \name → putStrLn $ "Hello, " <> name
```

What is `>>=`?

```
class Monad m where
  (>>=) :: m a → (a → m b) → m b
  -- ...
```

The monadic **bind** combinator `>>=` sequentially composes effectful computations with a **data dependency**:

in the above example you need the actual result of `getLine` to produce the `putStrLn "Hello, username"` computation!

Monadic tagless final DSL

Say we have a logging construct, and it is implemented using monads in the real interpreter.

```
class Monad repr => Logging repr where
    log :: String → repr ()
```

Of course we can implement `Logging` for `Print`..

```
instance Logging Print where
    log x = Print $ const $ "log \"" <> x <> "\""
```

Monadic tagless final DSL

Say we have a logging construct, and it is implemented using monads in the real interpreter.

```
class Monad repr => Logging repr where
    log :: String → repr ()
```

Of course we can implement `Logging` for `Print`..

```
instance Logging Print where
    log x = Print $ const $ "log \"" <> x <> "\""
```

But `Logging` also requires `Monad`.

```
instance Monad Print where
    l >>= r = Print $ \c → unPrint l c <> " >>= " <>
                             (unPrint r ??undefined??) c
```

We do not have the actual result of the left computation – `Print` is a **symbolic** interpreter!

Not-necessarily-monadic tagless final DSL

We can't use monads! But wait, all is not lost..

- we do not actually **need** monads in the syntactic typeclasses.

Not-necessarily-monadic tagless final DSL

We can't use monads! But wait, all is not lost..

- we do not actually **need** monads in the syntactic typeclasses.
- we need monadic composition in the evaluator, but regular functional composition in the printer.

Not-necessarily-monadic tagless final DSL

We can't use monads! But wait, all is not lost..

- we do not actually **need** monads in the syntactic typeclasses.
- we need monadic composition in the evaluator, but regular functional composition in the printer.
- solution: abstract over the composition method!

Not-necessarily-monadic tagless final DSL

We can't use monads! But wait, all is not lost..

- we do not actually **need** monads in the syntactic typeclasses.
- we need monadic composition in the evaluator, but regular functional composition in the printer.
- solution: abstract over the composition method!

```
class RCombinators repr where
  obind :: repr a → (repr a → repr b) → repr b

instance RCombinators Eval where
    a `obind` f = (>>=) a $ f . return

instance RCombinators Print where
  a `obind` f = Print $ \c → unPrint l c <> " >>= " <> unPrint r c
```

Not-necessarily-monadic tagless final DSL, pt. 2

Now we have to use the `RebindableSyntax` language extension to make `do` notation work with our custom combinator.

```
prog = let (>>=) = obind in do
    ident ← generateId
    log ident
```

Not-necessarily-monadic tagless final DSL, pt. 2

Now we have to use the `RebindableSyntax` language extension to make `do` notation work with our custom combinator.

```
prog = let (>>=) = obind in do
    ident ← generateId
    log ident
```

Actually, we have to repeat the process for `(>>)` to make `do` lines that don't bind variables work.

Not-necessarily-monadic tagless final DSL, pt. 2

Now we have to use the `RebindableSyntax` language extension to make `do` notation work with our custom combinator.

```
prog = let (>>=) = obind in do
    ident ← generateId
    log ident
```

Actually, we have to repeat the process for `(>>)` to make `do` lines that don't bind variables work.

Actually, in our production DSL, we overload a lot more things this way, because we want to allow lots of operations inside the DSL..

Combining EDSL Code and Regular Code in One Module

- `RebindableSyntax` uses definitions from the current **scope**.
- `let` / `where` bindings are the easiest way to bring definitions into local scope..

Combining EDSL Code and Regular Code in One Module

- `RebindableSyntax` uses definitions from the current **scope**.
- `let` / `where` bindings are the easiest way to bring definitions into local scope..
- but repeating this for lots of functions is very cumbersome:
    - `let (>>=) = obind; (>>) = oseq; ifThenElse = ite; (+) = add; .. in ..`

Combining EDSL Code and Regular Code in One Module

- `RebindableSyntax` uses definitions from the current **scope**.
- `let` / `where` bindings are the easiest way to bring definitions into local scope..
- but repeating this for lots of functions is very cumbersome:
    - `let (>>=) = obind; (>>) = oseq; ifThenElse = ite; (+) = add; .. in ..`
- importing a module with all the definitions is easy, but confines EDSL programs to their own modules;

Combining EDSL Code and Regular Code in One Module

- `RebindableSyntax` uses definitions from the current **scope**.
- `let` / `where` bindings are the easiest way to bring definitions into local scope..
- but repeating this for lots of functions is very cumbersome:
  - `let (>>=) = obind; (>>) = oseq; ifThenElse = ite; (+) = add; .. in ..`
- importing a module with all the definitions is easy, but confines EDSL programs to their own modules;
- would be nice to be able to bring the whole EDSL into scope using `RecordWildCards`: `let DSL{..} = ourDsl in ..`

Combining EDSL Code and Regular Code in One Module, pt. 2

To make `let DSL{..} = ourDsl in ..` work, we need to define these things:

```
noDsl = DSL { (==) = (Prelude.==)
            , (>>=) = (Prelude.>>=) }
dsl = DSL { (==) = eq
          , (>>=) = obind }
```

What type would DSL have? Well, we need to **decide** between overloaded types and standard library types..

Combining EDSL Code and Regular Code in One Module, pt. 2

To make `let DSL{..} = ourDsl in ..` work, we need to define these things:

```
noDsl = DSL { (==) = (Prelude.==)
            , (>>=) = (Prelude.>>=) }
dsl = DSL { (==) = eq
          , (>>=) = obind }
```

What type would DSL have? Well, we need to **decide** between overloaded types and standard library types..

We need functions at the **type level**! These are called type families.

Here's how to decide between wrapped and unwrapped values based on a boolean:

```
type family W b repr a where
  W 'False repr a = a
  W 'True  repr a = repr a
```

Similarly, in the paper we define `WM` for choosing between two wrappings, and `WMC` for choosing between a `Monad m` constraint and an empty constraint.

Combining EDSL Code and Regular Code in One Module, pt. 3

The type of `DSL` uses these type families to decide between DSL and regular functions:

```
data DSL w repr a b m = DSL
  { (==) :: Eq a => W w repr a → W w repr a → W w repr Bool
  , (>>=) :: WMC w m
          => WM w repr m a → (W w repr a → WM w repr m b)
           → WM w repr m b }
```

Combining EDSL Code and Regular Code in One Module, pt. 3

The type of `DSL` uses these type families to decide between DSL and regular functions:

```
data DSL w repr a b m = DSL
  { (==) :: Eq a => W w repr a → W w repr a → W w repr Bool
  , (>>=) :: WMC w m
          => WM w repr m a → (W w repr a → WM w repr m b)
           → WM w repr m b }
```

The types of the aforementioned `DSL` values are:

```
noDsl :: DSL 'False (Const Void) a
dsl :: RCombinators repr => DSL 'True repr a b m
```

Now, when the module with the DSL type is imported, we have to decide between using regular and DSL operations with a `let DSL{..} = ..` construct.

Thank you!

- *Carette J.*, *Kiselyov O.*, *Shan C.-c.* — Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. — // J. Funct. Program. — USA, 2009. — Sept. — Vol. 19, no. 5. — P. 509–543.
- The expression problem. /. — P. Wadler [et al.] // Posted on the Java Genericity mailing list. — 1998.