Techniques for Implementation of Symbolically Interpretable Haskell EDSLs

Grigoriy Volkov
National Research University Higher School of Economics
Institute for System Programming of the Russian Academy of Sciences
Moscow, Russia
Email: gdvolkov@ispras.ru

***Abstract*—Embedded domain-specific languages (EDSLs) are often used in typed functional programming languages like Haskell to implement executable formal specifications. A common requirement for the specification language is to be symbolically interpretable, which makes it possible to export specifications into external formats. Executable formal specifications often require abilities like sequential composition with data dependencies. Such abilities require powerful expressive abstractions (e.g. monads) which cannot be interpreted symbolically in the host language. In this paper we present various implementation techniques that allow symbolic interpretation of DSLs that require such abstractions in some practical cases.**

***Index Terms*—functional programming, Haskell, domain-specific languages, symbolic interpretation**

I. Introduction

Haskell is a typed functional programming language famous for, among other things, hosting lots of embedded domain-specific languages (EDSLs). There are EDSLs for everything from concurrent data access[1] to safe systems programming[2] (compiling to C code) and hardware description[3] (compiling to synthesizable VHDL/Verilog code). The Haskell language ecosystem offers many powerful tools, compiler features and conventions for building embedded languages that assign custom semantics to Haskell expressions.

At ISP RAS, we are developing formal specification languages embedded in Haskell. These languages have to comply with some unique requirements. Most notably, we need to be able to interpret programs in these languages symbolically, without running the program on actual data – for example, one such interpretation is printing the syntax tree to a string resembling the original source code. A more useful but similar usage would be to export the specification into a format suitable for various external tools. At the same time, these languages are designed to describe sequences of events and state operations, which means that expressions must be composed sequentially, in strict order – with data dependencies between expressions. This requires the usage of monads for interpretation on actual data. Also, these languages contain many of the usual standard library operators (equality, comparison) and control flow constructs (if-then-else), which are required be symbolically executable as well.

The combination of these requirements forms a class of EDSLs that are difficult to implement with standard techniques. The use of monads is in direct conflict with symbolic interpretation, as the signature of the monadic bind combinator makes it impossible to keep values always wrapped inside interpreter-provided types, and the use of standard operators requires them to be overloadable. However, the flexibility of Haskell (with GHC extensions) allows us to work around problems like this one.

In this paper we review the capabilities of Haskell for building EDSLs, describe our experience with implementing EDSLs with the aforementioned properties, and propose novel methods for dealing with problems that arise when dealing with the described class of EDSLs.

The outline of this paper is as follows: in Section II, various classifications of EDSLs are examined; in Section III, the problems discovered from our experience with building imperative, symbolically interpretable EDSLs are described and corresponding solutions are proposed; in Section IV, a conclusion is provided along with directions for future research.

II. Fundamentals of Haskell EDSLs

Embedded DSLs are, in essence, libraries that offer a limited set of types and functions with certain rules for composing expressions out of them[4].

One traditional classification of EDSLs is shallow versus deep embedding[5]: shallow embedding uses semantics of the host language directly, while deep embedding preserves the syntax of the embedded language, constructing an abstract syntax tree — typically, using algebraic data types (ADTs).

In [6], a classification of two encodings of embedded languages was introduced: initial and final encoding. Initial encoding means the abstract syntax tree (AST) of a program in the embedded language is explicitly represented using ADTs. The naive version of this is called tagged, because tagged sum types are used. The usage of generalized algebraic data types (GADTs) allows for a tagless

initial encoding. The *tagless final encoding* is different: instead of data constructors, overloaded functions are used to construct expressions in the embedded language. This is made possible by Haskell's typeclass feature, which allows for ad-hoc polymorphism. (Typeclasses are roughly similar to interfaces in object-oriented languages like Java, except typeclass instances are defined independently of the type, and they can be multi-parameter.) Despite not using ADTs, the tagless final encoding is still related to deep embedding: initial and final encodings are isomorphic, and it is trivial to construct a tagless final interpreter that would construct an explicit AST.

The major benefit added by the tagless style is extensibility: unlike the set of data constructors of a sum type[7][1], the set of Haskell functions is open for extension, so a library that provides a tagless final EDSL can be extended by consumers of the library (new expressions can be added to the language). In a final EDSL, the typeclasses define syntax of the language, while their instances define particular interpretations of that syntax, i.e. their semantics. The values that represent programs in the embedded language are polymorphic, parameterized by the type of the interpreter.

An interesting consequence of that is that interpreters with complex rules for nesting expressions can use different types for different "sublanguages". For example, in imperative programing languages a 'break' statement only makes sense inside of a loop. When implementing this in a tagless final EDSL, the 'break' statement would be defined in a separate typeclass, and the evaluating interpreter would only provide instances of it for types that represent expressions inside the loop body.

Another orthogonal classification of EDSLs is by which mechanism is used for variable bindings. One option is de Brujin indices[10]: explicit tracking of the environment, with numeric indices used to refer to variables. Another is higher-order abstract syntax[11]: usage of the host language's lambda expressions.

III. Discovered Problems and Solutions

We have been building an embedded DSL for software behavior specification that is made to resemble imperative programming: a specification describes a sequence of events, state changes and logical checks that might cause the specification checker (evaluator) to abort early. Therefore, the implementation of the evaluator in Haskell requires monads, and in fact various aspects of the evaluator map quite naturally onto common monads and monad transformers (such as ExceptT for early exit).

Our DSL is built in the tagless final style with higher-order abstract syntax. Papers related to the tagless final style[6][12] do not discuss embedding imperative languages, but the integration of monads into tagless final was straightforward. However, it has made symbolic interpretation impossible. In this chapter, we describe our solution to that problem, as well as some additional problems and solutions.

A. Symbolic Interpretation of Expressions with Monadic Composition

Monads are the fundamental abstraction in the Haskell standard library for dealing with sequential composition. The monadic bind operator `(>>=) :: m a -> (a -> m b) -> m b` composes two computations where the second one has a *data dependency* on the first one, which means that it requires the first one to be executed first: it is impossible to construct the computation `m b` without unwrapping the `a` from `m a`. Monads are obviously important for imperative programming, and composing I/O actions this way[2] is one of the first things Haskell beginners usually learn.

Unfortunately, the usage of monadic composition presents a serious problem for any kind of symbolic interpretation such as printing. As described above, the signature of the bind operator requires the interpreter to have access to actual data – an unwrapped value of the return type. In a symbolic interpreter, such values do not exist. The type of expressions defined for such an interpreter looks like, for example, `newtype Print a = { unPrint :: String }`. Note that the type variable 'a', which is the raw type of the expression (a number, string, list, etc) is a phantom type variable, i.e. it is thrown away on the right side – the symbolic interpreter only has a symbolic representation of the expression, such as a string in the case of printing. This means that for printing and similar interpreters, we cannot use monads at all, even though we need to use them for the interpreter that does real evaluation.

However, it is possible to only use monadic composition in one interpreter and functional composition in another, thanks to the RebindableSyntax GHC extension. We can define a typeclass that uses functional composition in the general case, and defaults to monadic composition for interpreters that are monadic:

```
class RCombinators repr where
  obind :: repr a
        -> (repr a -> repr b)
        -> repr b
  oseq  :: repr a -> repr b -> repr b
  default obind :: Monad repr
                => repr a
                -> (repr a -> repr b)
                -> repr b
  a `obind` f =
    (Prelude.>>=) a \$ f . return
  default oseq :: Monad repr
               => repr a
```

[1]Compositional data types[8][9] can be used as an alternative, however they are quite difficult to use

[2]Often with do notation instead of direct usage of the bind operator

```
              -> repr b
              -> repr b
  oseq = (Prelude.>>)
```

And thanks to RebindableSyntax, we can set `(>>=) = obind` and `(>>) = oseq`. This works with do notation just fine.

Now we can define non-monadic instances of this class for symbolic interpreters like printers:

```
instance RCombinators Print where
  Print a `obind` f =
    Print $ "let" <+> "VAR" <+> "=" <+>
      parens a <+> "in" <$>
      parens (unPrint $ f $ Print "VAR")
  Print a `oseq` Print b =
    Print $ a <+> ">>" <$> b
```

(This example uses a pretty-printing library, but does not use a numbering scheme for lambda arguments, just for demonstration.)

Here you can see that because the typeclass is defined using functional, not monadic composition, we are not required to provide actual unwrapped values (which do not exist), and we provide a symbolic representation (here, Print "VAR") which is of the wrapped type.

B. Combining EDSL Code and Regular Code in One Module

The RebindableSyntax GHC extension tells GHC to use in-scope definitions (instead of hard-coded standard library ones) for common operators including composition (`>>=`, `>>`), comparisons (`==`, `>=`, and so on) and even if-then-else blocks. This allows an advanced EDSL to reuse regular Haskell syntax for custom EDSL expressions. However, manually importing all the EDSL operators into the current scope for each top-level definition of an EDSL expression is tedious and distracting, while importing them into module scope prevents regular, non-DSL code in the same module from functioning normally. The latter may very well be an acceptable trade-off, but we have found a solution that allows mixing EDSL and regular code in one module without too much syntactic noise.

The key is to use the RecordWildCards GHC extension, which allows for a lightweight and short expression to be used for bringing multiple definitions into scope at once. The regular Haskell functions would start with `let DSL{..} = noDsl in (code)` and EDSL functions with `let DSL{..} = ourDsl in (code)`. (Note: literally two dots, not something omitted from the article.) The DSL type would be a collection of operators supported by RebindableSyntax. Type-level functions (Type Families) would be used to provide choice between EDSL and normal types. We have discovered three common cases, each of which requires its own type family. The first is the choice between raw and wrapped values:

```
type family W b (repr :: * -> *) a where
  W 'False repr a = a
  W 'True  repr a = repr a
```

The second is the choice between two wrapped values (i.e. wrapped in our EDSL representation or an arbitrary functor):

```
type family WM b (repr :: * -> *)
                 (m :: * -> *) a where
  WM 'False repr m a = m a
  WM 'True  repr m a = repr a
```

The third is for type constraints related to the second case. When choosing between the bind operator (»=) for arbitrary monads and for EDSL expressions, we need to choose between the Monad typeclass constraint and no typeclass constraint:

```
type family WMC b (m :: * -> *) where
  WMC 'False m = Monad m
  WMC 'True  m = ()
```

These type families use a type-level boolean argument for disambiguation – if we try to match on the provided wrapper type, GHC is not able to infer types for the operators. The DSL type uses these type families like so:

```
data DSL w repr a b m = DSL
  { ifThenElse :: W w repr Bool
               -> W w repr a
               -> W w repr a
               -> W w repr a
  , (==) :: 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 }
```

To define a value of this type that corresponds to standard functions, we set the type variable 'w' of the DSL type to false (and the 'repr', which is the DSL wrapper variable, to Const Void, which is a functor that has no values), which causes our type families to calculate types equal to standard expressions on normal unwrapped values. Then we can use standard functions from the Prelude (and a custom but obvious if-then-else definition) for the fields:

```
noDsl :: DSL 'False (Const Void) a
noDsl = DSL { ifThenElse = ite
            , (==) = (Prelude.==)
            , (>>=) = (Prelude.>>=) }
  where ite True a \_ = a
        ite False \_ b = b
```

And to define a value of this type that imports the operators for our embedded language, we set 'w' to true, which

causes our type families to resolve to wrapped types, and we provide a type variable of our own for the 'repr', with the constraints corresponding to EDSL typeclasses placed on it. We can use EDSL functions then to define the operators:

```
dsl :: (RLogic repr, RCombinators repr)
    => DSL 'True repr a b m
dsl = DSL { ifThenElse = if_
          , (==) = eq
          , (>>=) = obind }
```

With all this code, we have successfully convinced GHC to allow us to use very short RecordWildCards based inclusions of fields from this structure for including operators with very different types.

IV. Conclusions and Future Work

In this paper, the problems of implementing symbolically interpretable Haskell EDSLs that require many unusual features (monadic composition, standard operators) were described, and several techniques for solving these problems were proposed. These techniques are currently used in the development of embedded formal specification languages for software testing tools at ISP RAS.

One direction for future research is development of techniques for manipulating user-defined data types inside symbolically interpretable EDSLs. Another is development of alternatives to the RebindableSyntax approach. (The experimental `overloaded` library recently appeared on Hackage, it looks like an interesting direction.)

References

[1] S. Marlow, L. Brandy, J. Coens, and J. Purdy, "There is no fork: An abstraction for efficient, concurrent, and concise data access," in *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '14, Gothenburg, Sweden: Association for Computing Machinery, 2014, pp. 325–337, ISBN: 9781450328739. DOI: 10.1145/2628136.2628144. [Online]. Available: https://doi.org/10.1145/2628136.2628144.

[2] T. Elliott, L. Pike, S. Winwood, P. Hickey, J. Bielman, J. Sharp, E. Seidel, and J. Launchbury, "Guilt free Ivory," in *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, ser. Haskell '15, Vancouver, BC, Canada: Association for Computing Machinery, 2015, pp. 189–200, ISBN: 9781450338080. DOI: 10.1145/2804302.2804318. [Online]. Available: https://doi.org/10.1145/2804302.2804318.

[3] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards, "Cλash: Structural descriptions of synchronous hardware using haskell," in *Proceedings of the 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, ser. DSD '10, USA: IEEE Computer Society, 2010, pp. 714–721, ISBN: 9780769541716. DOI: 10.1109/DSD.2010.21. [Online]. Available: https://doi.org/10.1109/DSD.2010.21.

[4] A. Löh, "Haskell for (E)DSLs," *Functional Programming eXchange*, 2012. [Online]. Available: https://www.andres-loeh.de/HaskellForDSLs.pdf.

[5] J. Gibbons and N. Wu, "Folding domain-specific languages: Deep and shallow embeddings (functional pearl)," in *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '14, Gothenburg, Sweden: Association for Computing Machinery, 2014, pp. 339–347, ISBN: 9781450328739. DOI: 10.1145/2628136.2628138. [Online]. Available: https://doi.org/10.1145/2628136.2628138.

[6] J. Carette, O. Kiselyov, and C.-c. Shan, "Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages," *J. Funct. Program.*, vol. 19, no. 5, pp. 509–543, Sep. 2009, ISSN: 0956-7968. DOI: 10.1017/S0956796809007205. [Online]. Available: https://doi.org/10.1017/S0956796809007205.

[7] P. Wadler *et al.*, "The expression problem," *Posted on the Java Genericity mailing list*, 1998. [Online]. Available: http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt.

[8] W. Swierstra, "Data types à la carte," *J. Funct. Program.*, vol. 18, no. 4, pp. 423–436, Jul. 2008, ISSN: 0956-7968. DOI: 10.1017/S0956796808006758. [Online]. Available: https://doi.org/10.1017/S0956796808006758.

[9] P. Bahr and T. Hvitved, "Compositional data types," in *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming*, ser. WGP '11, Tokyo, Japan: Association for Computing Machinery, 2011, pp. 83–94, ISBN: 9781450308618. DOI: 10.1145/2036918.2036930. [Online]. Available: https://doi.org/10.1145/2036918.2036930.

[10] N. G. De Bruijn, "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem," in *Indagationes Mathematicae (Proceedings)*, North-Holland, vol. 75, 1972, pp. 381–392.

[11] F. Pfenning and C. Elliott, "Higher-order abstract syntax," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, ser. PLDI '88, Atlanta, Georgia, USA: Association for Computing Machinery, 1988, pp. 199–208, ISBN: 0897912691. DOI: 10.1145/53990.

54010. [Online]. Available: https://doi.org/10.1145/53990.54010.

[12] O. Kiselyov, "Typed tagless final interpreters," in *Proceedings of the 2010 International Spring School Conference on Generic and Indexed Programming*, ser. SSGIP'10, Oxford, UK: Springer-Verlag, 2010, pp. 130–174, ISBN: 9783642322013. DOI: 10.1007/978-3-642-32202-0_3. [Online]. Available: https://doi.org/10.1007/978-3-642-32202-0_3.