Modular Executable Language Specifications in Haskell

Mirjana Ivanović
mira@unsim.ns.ac.yu
Viktor Kunčak
viktor@uns.ns.ac.yu
Institute of Mathematics, Novi Sad

Abstract

We propose a framework for specification of programming language semantics, abstract and concrete syntax, and lexical structure. The framework is based on Modular Monadic Semantics and allows independent specification of various language features. Features such as arithmetics, conditionals, exceptions, state and nondeterminism can be freely combined into working interpreters, facilitating experiments in language design. A prototype implementation of this system in Haskell is described and possibilities for more sophisticated interpreter generator are outlined.

1 Introduction

Denotational Semantics is a widely used method for formal specification of programming language semantics. It is a complete semantics, which permits proving all relevant program properties, and also enables automatic generation of language interpreters from language specifications. One of the problems which hinder wider use of Denotational Semantics is its *lack of modularity*. Within last decade an approach called *Modular Monadic Semantics* was proposed as a means to structure Denotational Semantics and make it more usable. This approach has theoretical advantages in systematic treatment of language features ([14]), but is also of great value for generating more efficient interpreters from specifications ([12], [4]).

In this paper we explore the benefits of using Modular Monadic Semantics for writing language specifications in Haskell. Unlike previous works, which focused on semantics ([12], [4]) or abstract syntax ([3]), we also pay special attention to modularity of the concrete syntax and lexical structure (henceforth termed *lexics*). The result is modularity of the interpreter along two dimensions: interpreter stages and language features.

We use higher-order, non-strict, purely functional programming language Haskell as implementation language. In addition to features present in Haskell98 ([8]), we adopt the use of multiparameter type classes, which are present in Hugs ([9]) and GHC (Glasgow Haskell Compiler) Haskell implementations and are likely to be incorporated into future language versions ([10]). We also use overlapping class instances, mostly to introduce the subtyping relation.

The rest of the paper is organized as follows. In Section 2 we explain our treatment of modularity. In Section 3 we use examples to illustrate language specification in our system. Section 4 explains the present implementation of the system in Haskell. Section 5 summarizes the advantages and disadvantages of the approach. Section 6 outlines a future implementation based on program generation and Section 7 concludes.

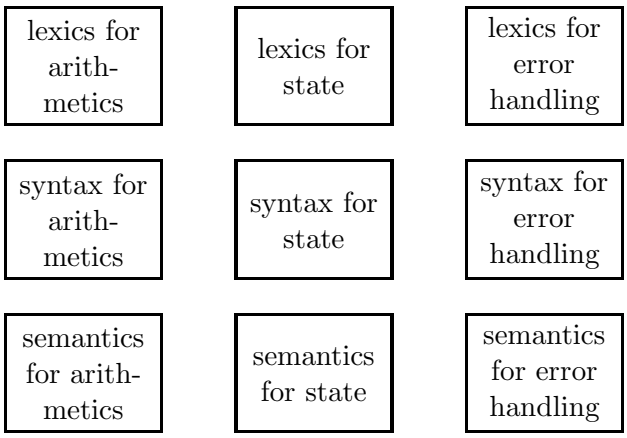

Figure 1: Two levels of modularity

2 Modularity in language design

2.1 Stages of interpreter

Division of interpreters and compilers into *stages* of lexical analysis, syntax analysis and semantics analysis is a widely used approach for handling the complexity of language processors ([1], [2]). This is a well known instance of modular design which allows using the most appropriate techniques and tools for each stage.

This approach also permits the reuse of stages of an interpreter or compiler, practical advantages of which are most evident in retargeting a programming language implementation for different platforms.

2.2 Another dimension of modularity

In this paper we focus on another level of modularity: division of an interpreter with respect to *language features*. The idea is to build blocks for features such as arithmetics, conditionals, loops, control flow, error handling, local definitions, and nondeterminism. By combining appropriate building blocks, an interpreter for the desired language is obtained.

Our intention is to applies 2-dimensional modularity: Grouping components along one dimension yields the usual stages of interpreter, whereas grouping them along the other dimension results in specifications of language features. We believe that this approach leads to a more systematic language design by providing foundations for the well recognized principle of orthogonality.

In both these dimensions of modularity, using Haskell as implementation language is advantageous. Lazy intermediate data structures help define clean interface between stages of interpretation, and demand-driven evaluation strategy makes the operational behavior of program identical to behavior of monolithic interpreter. The advantages are even more striking in the case of feature-wise modularity. We know of no other language apart from Haskell and its variant Gofer which have been able to capture the idea of Modular Monadic Semantics with such degree of precision. While the advantage of using a purely functional programming language for specification of Denotational Semantics is evident, it is the use of constructor (multiparameter) type classes that made it possible to express this new dimension of modularity in the type system, without the need for

program generation.

3 Modular language specification in Haskell

We have built a set of modules in Haskell which enable modular specification of a programming language and its interpreter. Using higher-order functions from these libraries, Haskell modules can be written that concisely describe interpretation stages for each language feature. We shall use arithmetics feature to illustrate the nature of these specifications and defer the details of implemented system to Section 4.

3.1 Lexics

The lexics of a language feature contains

- token data type definition (here: `Token`)
- list (`lexemes`) of lexeme specifications, which are pairs of
  - regular expression
  - function of type `[Char] -> Token`

```
>data Token = N Int | Plus | Minus | Times | Divided | Power
>lexemes = [(pInt, makeInt),
>           (rSym '+', \_ -> Plus),  (rSym '-', \_ -> Minus),
>           (rSym '*', \_ -> Times), (rSym '/', \_ -> Divided),
>           (rSym '^', \_ -> Power)]
>pInt = digit <&> (rMany digit)        -- digit digit*
>  where digit = rAnyOf "0123456789"
>makeInt ds = N (foldl op 0 ds)
>  where op n d = 10*n + (ord d - ord '0')
```

Token data type defines the interface between lexical analysis and syntax analysis stage for the particular language feature. Each regular expression defines a sequence of characters comprising a lexeme of the language. Regular expressions are built using operators `<&>`, `<|>`, and `rMany` which correspond to concatenation, alternative, and iteration, respectively. The function which forms the second component of the pair in lexeme specification maps the lexeme into its `Token` representation, which is used in syntax analysis stage.

3.2 Syntax

Specification of the syntax for a language feature contains

- abstract syntax tree definition
- a function (`par`) mapping a token to operator description

```
>data Tree x = Const Int
>            | Add x x | Sub x x
>            | Mul x x | Div x x
```

```
>            | Pow x x
>par (N x)   = literal  (Const x)
>par Plus    = infixOpL 502 Add
>par Minus   = infixOpL 502 Sub
>par Times   = infixOpL 503 Mul
>par Divided = infixOpL 503 Div
>par Power   = infixOpR 504 Pow
```

Declaration of the abstract syntax tree contains a type variable in place of recursion. Declaring the tree as constructor rather than a type is essential for modularity of abstract syntax, as we shall see in 4.2. Specification of the operator corresponding to a token is achieved using predefined higher-order functions `literal`, `infixOp`, `infixOpL`, `infixOpR`, `prefixOp` and others. For instance, `infixOpL` function defines a left-associative infix binary operator with given priority and syntax tree constructor. `infixOpR` can be used to specify right-associative operators, `prefixOp` and `prefixBinOp` for prefix operators, and `ternary` for ternary prefix operator. The library can easily be extended with new operator specifications.

3.3 Semantics

Specification of semantics for a language feature is based on Modular Monadic Semantics. In Haskell it is expressed in the form of `Algebra` instance declaration.

```
>instance (Subtype Int v, ErrMonad String m) => Algebra Tree (m v) where
>  phi e = case e of
>            (Const x)    -> returnInj x
>            (Add xm ym)  -> lift2sub plus xm ym
>            (Sub xm ym)  -> lift2sub minu xm ym
>            (Mul xm ym)  -> lift2sub tims xm ym
>            (Div xm ym)  -> do x <- mprj xm
>                               y <- mprj ym
>                               if y==0 then eThrow "Division by zero"
>                                       else returnInj (divi x y)
>            (Pow xm ym)  -> lift2sub pow xm ym
```

The type constructor `Tree` can be treated as a signature of an algebra. This instance declaration defines a particular algebra of the signature `Tree` by interpreting operations represented as nodes of the abstract syntax tree ([3]). This interpretation corresponds to semantics definitions in Denotational Semantics ([16]), but adopts the use of Modular Monadic Semantics to achieve a higher level of abstraction and modularity.

Modular Monadic Semantics is based on the notion of *monad* ([13], [14], [18]). As far as programming in Haskell is concerned, monad is a higher-order abstract data type, given by a class declaration

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

and satisfying the following laws:

```
m >>= return = m
(return x) >>= f = f x
m >>= (\a -> (f a >>= g)) = (m >>= f) >>= g
```

Monads generalize function application, a fact formalized by the following instance declaration.

```
>newtype Id x = Id { unId :: x }
>instance Monad Id where
>  return = Id
>  (Id x) >>= f = f x
```

Programs expressed via monad composition become easier to maintain, since we can tune the meaning of the program by changing the underlying monad. To make the use of monads more convenient, Haskell provides syntactic sugar in the form of `do`-notation, whose essence is given by the following equation.

```
do {x <- m; e} = m >>= (\x -> e)
```

In Modular Monadic Semantics the domain of interpretation is decomposed into *computation* type constructor `m`, and *value* type `v`. The domain `d` is then `d = m v`. The computation constructor is a subclass of `Monad` class, supporting additional operations which are needed to interpret particular language feature. For instance, the `ErrMonad` is defined by

```
>class Monad m => ErrMonad e m where
>  eThrow :: e -> m a
```

which permits the use of `eThrow` in interpretation of the `Div` tree node. Similarly, we use class constraints to allow any supertype of `Int` as the value `v`. By using sophisticated class mechanism of Haskell we can specify minimal requirements for the domain of interpretation, which is crucial for modularity.

3.4 Putting the components together

Here we outline the way in which components are composed into working interpreter. Looking back at figure 2.2, we first group horizontally components in each stage, using `clex` operator for composing lexical specifications, `cpar` for composing syntax, and monad transformers to create the final interpretation domain. The description of each stage is then turned into function, and these functions are composed in sequence to provide the final interpreter as a function `String -> String`. We give some more details on this process in 4.2.

4 A simple implementation

Current implementation of the system is a collection of Haskell modules. These modules implement abstract data types for the specification of lexics, syntax, and semantics of language components, functions for transforming specifications into executable functions, as well as higher-order functions for merging component descriptions into final language specification.

4.1 Implementing stages of interpreter

Implementation of lexical analysis is based on transformation of regular expressions into Nondeterministic Finite Automaton (NFA) and Deterministic Finite Automaton (DFA). In the first stage, regular expression is used to derive a NFA states of which are nodes of the regular expression tree ([1]). This avoids the creation of empty (ϵ) transitions. Furthermore, to avoid the potential explosion of states in DFA construction, lazy transition evaluation is applied to construct DFA states and transitions during the lexical analysis ([1]). This approach can be seen as a manual application of the memoization technique ([5]).

Implementation of (concrete) syntax analysis is based on precedence parsing. Operator specifications are evaluated into transitions of a state machine. The machine contains *operator stack* and *argument stack*. The parsing algorithm can be seen as a result of merging translation of expressions into postfix form ([17]), which uses the operator stack, and expression evaluation using argument stack. The algorithm is extended to provide handling of unary, binary and ternary infix expressions as well as error detection. The choice of precedence parsing may seem unusual in the light of limitations of this technique, but it turns out to be a convenient and efficient choice when modularity is imperative.

Implementation of semantics derives from the previous work on Modular Monadic Semantics. It uses *monad transformers* ([14], [12], [4]) for monad composition and *lifting* to merge the computation effects required by various language features. Our semantics library contains definitions for identity and nondeterminism monad as well as monad transformers for errors, environments, state, and continuations. Each transformer extends a monad with additional operations. For instance, the continuation monad transformer adds the operation `callcc` (call with current continuation).

```
class Monad m => ContMonad m where
  callcc :: ((a -> m b) -> m a) -> m a
newtype ContT c m a = Cont {unCont :: (a -> m c) -> m c}
instance Monad m => Monad (ContT c m) where
  return a = Cont (\k -> k a)
  (Cont m) >>= f = Cont (\k -> m (\a -> unCont (f a) k))
instance Monad m => ContMonad (ContT c m) where
  callcc f = Cont (\k -> unCont (f (\v -> Cont (\k' -> k v))) k)
```

The `ContMonad` class represents computations that require complex flow of control, such as (equivalents of) non-local jumps. The `newtype` declaration introduces a type constructor `ContT` which transforms its argument to enable implementation of functionality for the new class. This new functionality is defined in the second instance declaration. The first instance declaration makes sure that the new type still supports the functionality of monad class.

4.2 Combining specifications

Combining specifications of various interpreter stages is central to our approach of modularity. We begin by describing the composition of semantics specifications, since this was the original problem of modular interpreters.

The key problem in Modular Monadic Denotational semantics is that of correct definitions of liftings. Liftings ensure that monad operations introduced by one transformer remain available after subsequent application of further monad transformers. The first step in lifting is to associate with each monad transformer a `lift` function which transforms original monad values into new monad values.

```
class (Monad m, Monad (t m)) => MonadT t m where
  lift :: m a -> t m a
instance Monad m => MonadT (ContT c) m where
  lift = Cont . (>>=)
```

This function is enough to lift any operation not containing monad in the domain type, such as `eThrow`. The following declaration defines lifting of `eThrow` through arbitrary monad transformer, which means that applying any monad transformer `t` to an `ErrMonad` yields not only `Monad`, but also `ErrMonad`.

```
instance (ErrMonad e m, MonadT t m) => ErrMonad e (t m) where
  eThrow = lift . eThrow
```

Some other cases of lifting are more difficult. Providing the definition for these cases amounts to describing interaction between individual semantic features ([12]).

Modularity of abstract syntax is based on the notion of sum of algebras. Abstract syntax trees, defined as constructors, represent algebra signatures. The sum of algebras is defined using the `Sum` constructor.

```
>newtype Sum f g x = Sum {uSum :: Either (f x) (g x) }
```

After forming the sum of abstract trees of all components, the `Fix` constructor is applied to create the recursive structure. On the one hand this corresponds to initial algebra over algebra signature. On the other hand, it is the abstract syntax tree which provides an interface between the analysis of concrete syntax and semantics analysis.

```
>newtype Fix f = In {out :: f (Fix f)}
```

The interpretation is captured by a multiparameter class `Algebra`.

```
>class Functor f => Algebra f a where
>  phi :: f a -> a
```

Sum of algebras is defined in the natural way.

```
instance (Algebra f a, Algebra g a) => Algebra (Sum f g) a where
  phi (Sum (Left  ef)) = phi ef
  phi (Sum (Right eg)) = phi eg
```

Finally the notion of expression value is defined given the interpretation of algebra.

```
>eval :: Algebra f a => Fix f -> a
>eval (In e) = phi (fmap eval e)
```

Modularity of concrete syntax and lexical structure of interpreted language does not seem to have attracted much attention so far. The previous approaches we are aware of either used monolithic syntax analysis stage that generates abstract syntax trees, or applied parsing combinators to make a trivial extension to abstract syntax from previous paragraph. The first approach means giving up modularity of the whole interpreter, so we avoid it. The second approach requires excessive use of parentheses since concrete syntax is a direct translation of abstract syntax. What is more, the resulting parsers are inefficient due to intensive backtracking in combinator parsers. This is because the concrete expression grammar is not LL(1) and no left factoring is performed.

Our choice of precedence parsing results in efficient token-driven algorithm which works as a deterministic push-down automaton. We achieved the modularity using higher order functions. The type of functions such as `infixOpL` is not just a state transition (denoted by `ParsingItem b`), but a function from `a->b` to the transition.

```
infixOpL :: Int -> (b -> b -> a) -> (a -> b) -> ParsingItem b
```

The previous specification of `Plus` token can be written equivalently as

```
par Plus r = infixOpL 502 Add r
```

Combining two syntax definitions can be achieved by a single higher-order function `cpar` which modifies the `r` argument.

```
cpar :: (tokL -> (treeL a -> b) -> c) ->
        (tokR -> (treeR a -> b) -> c) ->
        Either tokL tokR -> (Sum treeL treeR a -> b) -> c
cpar parL parR = f
  where
    f (Left  x) r = parL x (r . Sum . Left)
    f (Right x) r = parR x (r . Sum . Right)
```

The combined parser accepts the sum of token types as a new token type and the sum of trees as new abstract tree type. By a simple map of lexeme specification lists we also achieve combination of lexical structure.

```
clex :: [Lexeme c tokL] -> [Lexeme c tokR] -> [Lexeme c (Either tokL tokR)]
clex lexL lexR = (map (m Left) lexL) ++ (map (m Right) lexR)
  where m f (exp,val) = (exp, f . val)
```

Higher-order functions `cpar` and `clex`, together with type constructors and instance declarations for modular abstract syntax and modular semantics enable the construction of an interpreter from the specifications of language features.

5 Results

We have implemented the framework for the specification of semantics, abstract syntax, concrete syntax and lexical structure of language components in Haskell, using Hugs98 interpreter and relying on multiparameter type classes and overlapping instances. This framework was then used to implement 8 language features: arithmetics, comparison relations and conditionals, environments (local names), exceptions (via continuations), (call by value) functions, loops, nondeterminism and state (assignable store). Using higher-order functions from the previous paragraph, all of these features can be combined into working interpreter.

The most immediate advantage of this approach is that we remain inside a general-purpose language Haskell, which yields more flexibility than special-purpose formalisms. We retain the convenience of static type-checking and the maintenance of specification is easier since there are no multiple program generation phases. Language feature specifications are first-order citizens, offering great potential for extensibility of the system. Using a language based on typed lambda calculus allows us to directly write Denotational Semantics definitions, fully integrated with syntax definitions. Type classes enable the desired modularity, making the specification easy to manage and

reason about. The system of library modules itself is small, and can be included into considerations of the interpreter semantics if needed.

The current implementation has also some drawbacks. Precedence parser is not flexible enough. The view of abstract syntax from [3] which we have adopted appears to be oversimplified. We may need the notion of multisorted algebras to capture the syntax of programming languages, unless we want to rely on more complex static semantics of the language. Lexical analyzer is not as efficient as we would like, and there is much space for improvements. Also, at this level of implementation the process of feature composition is not fully transparent to the user of our library modules. This is mainly due to the limitations of the type system and is evident when constructing recursive domains.

6 Future work

Two major areas for future work are extending the flexibility of specification and improving the efficiency of resulting interpreters. In our simple implementation we have often faced limitations of the type system, even in the presence of language extensions such as multiparameter type classes and overlapping instances. While we have demonstrated that much can be done using the type system as a metaprogramming tool, we think that some sort of program generation (metacomputation) approach will be necessary for future enhancements.

Our intention is that we keep as many benefits of the current system as possible as we move to program generation. For the start, we would not like to loose the safety of type checking. This is in contrast to most compiler-compiler tools that syntactically merge semantic actions with the generated code, deferring the consistency checks to target program compilation. The alternative we propose is extension to the Haskell language (or its relevant subset) with new constructs (syntactic sugar) for specification of syntax and lexical structure of programming languages. The implementation of this language would perform syntactic sugar elimination and limited form of partial evaluation to obtain (the representation of) an ordinary Haskell program. In this way we hope to keep the advantages of the current approach while improving the efficiency and flexibility.

Another potential line of work is compiler generation. The step from an interpreter to compiler is conceptually simple, but forms the body of work on compilation technology for several decades. Theoretical foundation of this are in specialization, partial evaluation and pass separation. Modular monadic semantics has been used for compilation in [11] and [6], but we are not aware of any system for compiler generation from specifications which would be based on this approach. A promissing possibility for practial application is the integration ([19]) of Modular Monadic Semantics with Action Semantics ([15]).

7 Conclusions

We have demonstrated that Haskell can successfully be used for the complex task of highly modular specification of programming language features. This approach allows fast creation of interpreter prototypes from their formal specifications, helping debug semantic definitions and providing a theoretical basis for future implementations. Using a general purpose language instead of specialized formalisms has many advantages. We believe that most of these can be retained in a future interpreter or compiler generator.

References

[1] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, Tools*. Adison Wesley, 1986.

[2] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[3] L. Duponcheel. *Using catamorphisms, subtypes and monad transformers for writing modular functional interpreters*, `http://cs.ruu.nl/people/luc`, 1995.

[4] D. Espinosa. *Semantic Lego*, PhD Thesis, Columbia University. `www.cs.columbia.edu`, 1995.

[5] A. J. Field, P. G. Harrison: *Functional Programming*, Addison-Wesley Publishers Ltd., 1988.

[6] W. L. Harrison, S. N. Kamin. Modular Compilers Based on Monad Transformers, *IEEE Computer Society International Conference on Computer Languages*, Loyola University, Chicago, 1998.

[7] J. Hughes. Why Functional Programming Matters. *Computer Journal 32(2)*, 1989.

[8] S. L. Peyton Jones, J. Hughes. *Haskell 98: A Non-strict, Purely Functional Language*. February 1999, language report available from `http://haskell.org/report`.

[9] M. P. Jones, J. C. Peterson. *Hugs98 User Manual, Revised version: September 1999*, `http://haskell.org/hugs`.

[10] S. L. Peyton Jones, M. P. Jones, E. Meijer, *Type Classes: An Exploration of the Design Space*, `www.cs.uu.nl/erik`.

[11] S. Liang. *Modular Monadic Semantics and Compilation*, PhD thesis, Yale University, 1998.

[12] S. Liang, P. Hudak. Modular denotational semantics for compiler construction. *ESOP'96: 6th European Symposium on Programming, Linkoping, Sweden*. Springer-Verlag, 1996.

[13] S. Mac Lane. *Categories for the Working Mathematician*, Springer-Verlag, 1971.

[14] E. Moggi. An abstract view of programming languages. Technical Report, ECS-LFCS-90-113, University of Edinburgh. `theory.doc.ic.ac.uk`, 1990.

[15] P. D. Mosses. A tutorial on Action Semantics. `www.brics.dk/pdm`, 1996.

[16] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, Cambridge, Massachusetts, and London, England, 1977.

[17] J. P. Tremblay, P. G. Sorenson. *The Theory and Practice of Compiler Writing*. McGraw-Hill Inc., 1985.

[18] P. Wadler. Monads for functional programming. In J. Jeuring, E. Meijer, eds.: *Advanced Functional Programming*, Proceedings of the Bastad Spring School, May 1995, Springer-Verlag Lecture Notes in Computer Science 925, 1995.

[19] K. Wansbrough. *A Modular Monadic Action Semantics*, Masters Thesis, University of Auckland, New Zeland, 1997.