NOVI SAD J. MATH.
VOL. 30, NO. 2, 2000, 19-27

Proceedings of the TARA 2000 Conference
Novi Sad, Yugoslavia, September 6-7, 2000

MODULAR LANGUAGE SPECIFICATIONS IN HASKELL

Mirjana Ivanović[1], Viktor Kunčak [1]

Abstract. A framework for specification of programming language semantics, abstract and concrete syntax, and lexical structure are proposed. 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.

AMS Mathematics Subject Classification (1991): 68N20

Key words and phrases: interpreter, modularity, Denotational Semantics

1. Introduction

Denotational Semantics is a widely used method for formal specification of the programming language semantics. It is a complete semantics, which permits proving all relevant program properties, and enables automatic generation of interpreters from the language specifications. One of the problems which hinders wider use of Denotational Semantics is its *lack of modularity.*In the last decade, an approach called *Modular Monadic Semantics* has been proposed as a means to structure Denotational Semantics and make it more usable. This approach has theoretical advantages in systematic treatment of language features ([9]) and is also of great value for generating more efficient interpreters from specifications ([8], [3]).

In this paper the benefits of using Modular Monadic Semantics for writing language specifications in Haskell are explored. Unlike previous works, which focused on semantics ([8], [3]), or abstract syntax ([2]), special attention is paid 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.

[1]Institute of Mathematics, University of Novi Sad, Trg Dositeja Obradovića 4, 21000 Novi Sad, Yugoslavia, {mira@unsim, viktor@uns}.ns.ac.yu

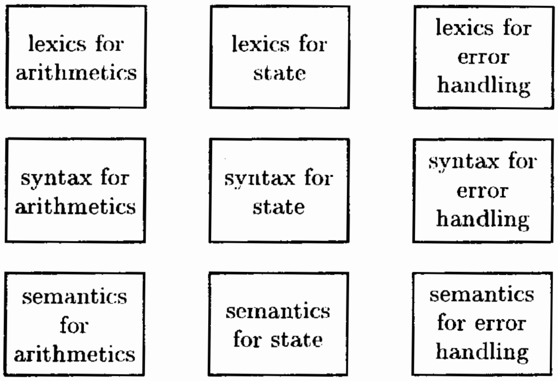
lexics for arithmetics
lexics for state
lexics for error handling
syntax for arithmetics
syntax for state
syntax for error handling
semantics for arithmetics
semantics for state
semantics for error handling

Figure 1: Two levels of modularity

The implementation is based on a higher-order, non-strict, purely functional programming language Haskell. In addition to features present in Haskell98 ([6]), multiparameter type classes were used. Overlapping class instances were also used, most notably in order to introduce the subtyping relation.

The rest of the paper is organized as follows. In Section 2, the treatment of modularity is clarified. In Section 3 some examples illustrating language specification in our system are presented. Section 4 explains the present implementation of the system in Hugs Haskell interpreter. Section 5 summarizes the results and Section 6 outlines a future implementation based on program generation.

2. Modularity in language design

Division of interpreters and compilers into *stages* of lexical analysis, syntax analysis and semantics analysis is a widely used approach whose benefits are well known. The focus of this paper is 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.

The intention is to apply a 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. It seems that this approach leads to a more systematic language design by providing foundations for the well recognized principle of orthogonality ([7]).

In both dimensions of modularity, using Haskell as implementation language is advantageous. Lazy intermediate data structures help define a clean interface between stages of interpretation, and demand-driven evaluation strategy makes the operational behavior of the program identical to the behavior of a monolithic interpreter. The advantages are even more striking in the case of feature-wise modularity. While the advantage of using a purely functional programming language for specification of Denotational Semantics is evident, the use of multiparameter type classes make it possible to express the new dimension of modularity in the type system, avoiding the program generation.

3. Modular language specification in Haskell

A set of modules in Haskell has been built which enables modular specification of the programming language and its interpreter. Using higher-order functions from these libraries, Haskell modules can be written to concisely describe interpretation the stages for each language feature. Arithmetics features will be used to illustrate the nature of these specifications.

Lexics of a language feature contains token data type definition (`Token`) and a list (`lexemes`) of lexeme specifications, which are pairs consisting of a regular expression and a function of the type `[Char] -> Token`.

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

Token data type defines the interface between lexical and syntax ~~analysis~~ 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 the 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 the syntax analysis stage.

Syntax specification for a language feature contains abstract syntax tree definition and a function (`par`) mapping a token to operator description.

```
data Tree x = Const Int | Add x x | Mul x x | Div x x | Pow x x
par (N x)    = literal (Const x)
par Plus     = infixOpL 502 Add; 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 a 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 the 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.

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

```
instance (Subtype Int v,ErrMonad String m)=>Alg Tree (m v) where
  phi e = case e of
    (Const x)     -> returnInj x
    (Add xm ym)   -> lift2sub plus xm ym
    (Mul xm ym)   -> lift2sub tims xm ym
    (Div xm ym)   -> do x <- mprj xm; y <- mprj ym
                        if y==0 then eThrow "Div 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 ([2]). This interpretation corresponds to the semantics definitions in Denotational Semantics ([11]), 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* ([9], [12]). As far as programming in Haskell is concerned, a 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 3 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
```

The programs expressed via monad composition become easier to maintain, as the meaning of the program can be tuned 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 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 a 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 the interpretation of the `Div` tree node. Similarly, class constraints are used to allow any supertype of `Int` as the value `v`. By using the sophisticated class mechanism of Haskell minimal requirements for the domain of interpretation can be specified, which is crucial for modularity.

Referring back to Figure 2, horizontal components in each stage are grouped first, 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 functions, and these functions are composed in sequence to provide the final interpreter as a function `String->String`.

4. Implementation issues

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 a 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, a regular expression is used to derive the NFA states of which are nodes of the regular expression tree ([1]). 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]).

Implementation of (concrete) syntax analysis is based on precedence parsing. The parsing algorithm can be seen as a result of merging the translation of expressions into postfix form, 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 precedence parsing turns out to be a convenient and effective choice when modularity is essential.

Implementation of semantics derives from the previous work on Modular Monadic Semantics. It uses ***monad transformers*** ([9], [8]) for monad composition and ***lifting*** to merge the computation effects required by various language features. The 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.

4.2. Combining specifications

Combining specifications of various interpreter stages is central in our approach to 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 the 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 the 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 an 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
```

There are more difficult cases of lifting. Providing the definition for these cases amounts to describing interaction between individual semantic features ([8]).

Modularity of abstract syntax is based on the notion of the 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 of the final abstract syntax tree.

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

The interpretation is captured by a multiparameter class `Alg` and the sum of algebras is defined in the natural way. Finally, the notion of expression value is defined given the interpretation of algebra.

```
class Functor f => Alg f a where phi :: f a -> a
instance (Alg f a, Alg g a) => Alg (Sum f g) a where
  phi (Sum (Left  ef)) = phi ef; phi (Sum (Right eg)) = phi eg
eval :: Alg f a => Fix f -> a
eval (In e) = phi (fmap eval e)
```

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

The choice of precedence parsing results in efficient token-driven algorithm which works as a deterministic push-down automaton. The modularity was achieved 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
```

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

```
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 a new abstract tree type. By a simple map of lexeme specification lists we also achieve combination of the lexical structure.

5. Results

The main result of the paper is the implementation of the framework for 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 applied 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 were combined into a working interpreter.

The most immediate advantage of this approach is the use of a general-purpose language Haskell, which offers more flexibility than special-purpose formalisms. Specifications are statically type-checked and the their maintenance is easier since there are no multiple program generation phases. Using a language based on typed lambda calculus allows immediate use of 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.

6. Conclusions and future work

The implemented framework demonstrates that Haskell can successfully be used for the complex task of highly modular programming language feature specification. 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, most of which can be retained in the future interpreter or compiler generator.

Two major areas for future work are extending the flexibility of specification and improving the efficiency of resulting interpreters. In this 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. This suggests that some sort of program generation (metacomputation) would be useful. The intention is to keep as many benefits of the current system as possible while turning to program generation approach. For the start, the safety of type checking should be retained. 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 propose here is an extension of 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 a limited form of partial evaluation to obtain an ordinary Haskell program.

Another potential line of work is compiler generation. The step from an interpreter to compiler is conceptually a simple one, but it has formed the body of work on compilation technology for several decades. Theoretical foundations of this are in specialization, partial evaluation and pass separation. Modular monadic semantics has been used for compilation in [8] and [4], 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 ([13]) of Modular Monadic Semantics with Action Semantics ([10]).

References

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

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

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

[4] Harrison, W.L., Kamin, S.N., Modular Compilers Based on Monad Transformers, IEEE Int. Conf. on Computer Languages, Loyola University, Chicago, 1998.

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

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

[7] Kunčak, V., Modular Interpreters in Haskell, BSc thesis, University of Novi Sad, 2000.

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

[9] Moggi, E., An abstract view of programming languages, Technical Report, ECS-LFCS-90-113, University of Edinburgh, 1990.

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

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

[12] Wadler, P., Monads for functional programming, In J. Jeuring, E. Meijer, eds.: Advanced Functional Programming, Proc. of the Bastad Spring School, May 1995, Springer-Verlag LNCS 925, 1995.

[13] Wansbrough, K., A Modular Monadic Action Semantics, MSc Thesis, University of Auckland, New Zeland, 1997.