Advanced programming concepts in a course on grammars and parsing

Johan Jeuring and Doaitse Swierstra
Department of Computer Science
Utrecht University, The Netherlands
{johanj,doaitse}@cs.uu.nl

Abstract

One of the important goals of the Computer Science curriculum at Utrecht University is to familiarize students with abstract programming concepts such as, for example, partial evaluation and deforestation. A course on grammars and parsing offers excellent possibilities for exemplifying and introducing such concepts. We have developed a course that exemplifies higher-order functions and lazy evaluation, and introduces abstract programming concepts such as partial evaluation, generic programming, deforestation, and abstract interpretation. This paper describes how we deal with these concepts in the course on grammars and parsing.

1 Introduction

One of the important goals of the Computer Science curriculum at Utrecht University is to familiarize students with abstract programming concepts such as, for example, partial evaluation and deforestation. Students should understand these concepts, recognize their applicability, and transform programs accordingly. The abstract programming concepts are introduced in a series of courses on programming.

- Functional programming
- Grammars and parsing
- Implementation of programming languages
- Software generation

In all of these courses the functional programming language Haskell is used to illustrate or implement programming concepts. After the first year introductory course on functional programming, the students take the second year course on grammars and parsing in which several important concepts of the introductory course are illustrated, and several new programming concepts are introduced. We started the development of the course on grammars and parsing in 1992, and have worked on the lecture notes and practicals since then. The material has reached a more or less stable state now. This paper describes how we exemplify and introduce the programming concepts.

This paper is organized as follows. Section 2 describes the structure of the course on grammars and parsing. Section 3 shows how we exemplify higher-order functions and lazy evaluation. Section 4 shows how we introduce some advanced programming concepts. Section 5 concludes.

2 Course contents

The course on grammars and parsing is a second year course with a four weeks working load. The required prior knowledge is taught in an introductory functional programming course. The course on grammars and parsing introduces, amongst others, the following topics.

- Context-free grammars
- Combinator parsing
- Grammar and parser design
- Regular languages and finite automata
- Compositionality and generic programming
- Attribute grammars
- LL(1) parsing

Some of these topics, such as context-free grammars and combinator parsing, reappear continuously throughout the course, some other topics, such as attribute grammars, are discussed briefly. The complete lecture notes can be obtained from the following web page (in Dutch):

`www.cs.uu.nl/docs/vakken/go/#literatuur`

The course on grammars and parsing is followed by courses on programming language implementation and software generation, in which several topics such as combinator parsing, compositionality and generic programming, reappear.

3 Higher-order functions and lazy evaluation

Higher-order functions such as `map`, `filter`, and `foldr` are treated in depth in the introductory course on functional programming. In the grammars and parsing course we use higher-order functions in almost all chapters. The most advanced uses of higher-order functions is found in the chapters on combinator parsing (based on [2]), compositionality, and attribute grammars. For example,

- In the chapter on combinator parsing, the first argument of the product parser <*> is a function that returns a function.

  ```
  type Parser s a  =  [s] -> [(a,[s])]

  symbol  ::  s -> Parser s s
  (<*>)   ::  Parser s (b -> a) ->
              Parser s b ->
              Parser s a
  (<+>)   ::  Parser s a ->
              Parser s a ->
              Parser s a
  ```

  Note that with the parser combinators we introduce a domain specific language.
- In the chapter on compositionality we define folds for various datatypes. A fold is a function that replaces constructors by argument functions, the typical example is

  ```
  foldr :: (a->b->b) -> b -> [a] -> b
  ```
- In the chapter on attribute grammars we construct, amongst others, a basic compiler for a small language. The compiler is written as a fold, which replaces constructors by functions that take the inherited attributes as arguments, and return the synthesized attributes.

We give several examples of lazy evaluation in the lecture notes, for example in the chapter on combinator parsers, and the chapter on attribute grammars, where we discuss amongst others the repmin problem.

4 Advanced programming concepts

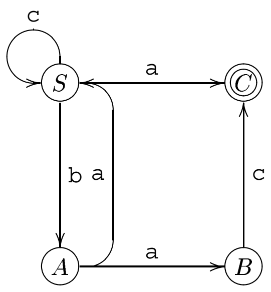
4.1 Partial evaluation

In the chapter on regular languages we give an implementation of both deterministic and nondeterministic finite state automata. We define the following class:

```
class NFA a b where
   start         :: [a]
     -- start states
   delta         :: b -> a -> [a]
     -- state transition function
   finals        :: [a]
     -- final states
   nfa           :: [b] -> [a]
     -- input processing function
   nfaAccept     :: [b] -> Bool
     -- checks if an input is accepted

   nfa  =  foldl (flip deltas) start
     where deltas a = union . map (delta a)
   nfaAccept xs  =
     intersect (nfa xs) finals /= []
```

where the definitions of union and intersect are omitted. Then we give an example instance of this class, and we show how the function nfa of such a concrete example can be partially evaluated by means of the finite bounded static variation method. For example, the following finite state machine:



is implemented by

```
data State   =  A | B | C | S
data Symbol  =  SA | SB | SC

instance NFA State Symbol where ...
```

where the, rather straightforward, instance definitions are omitted. For the function nfa in this instance we calculate:

```
  nfa [x1,x2,...,xn]
= foldl (flip deltas)
        start
        [x1,x2,...,xn]
= foldl (flip deltas)
        (deltas x1 start)
        [x2,...,xn]
= case x1 of
     SA -> foldl (flip deltas)
                 (deltas SA start)
                 [x2,...,xn]
     SB -> foldl (flip deltas)
                 (deltas SB start)
                 [x2,...,xn]
     SC -> foldl (flip deltas)
                 (deltas SC start)
                 [x2,...,xn]
```

All these equalities are simple transformation steps for functional programs. Note that the first argument of foldl is always flip deltas, and the second argument is one of the six sets of states [S], [A], [B], [C], [B,S], [C,S] (the possible results of deltas). Since there are only a finite number of results of deltas (six, to be precise), we can define a transition function for each state:

```
nfaS, nfaA, nfaB, nfaC, nfaBS, nfaCS
  ::  [Symbol] -> [State]
```

Each of these functions is a case expression over the possible input symbols. Note that by partially evaluating the function nfa we have obtained a function that is the implementation of the deterministic finite state automaton corresponding to the nondeterministic finite state automaton.

4.2 Generic programming

Since they are the standard recursion operators on (abstract syntax) trees, folds are used throughout the lecture notes. The chapter on compositionality introduces folds on several datatypes, and shows how they are used to process abstract syntax trees (values of datatypes). Folds are used for evaluating expressions, pretty printing trees, generating code, etc. For example, an abstract syntax tree of the grammar with the productions $S \rightarrow$ (S) $S \mid \epsilon$ can be represented by a value of the datatype Parentheses.

```
data Parentheses =
    Match Parentheses Parentheses
  | Empty
```

The fold on this datatype is defined as follows:

```
type ParenthesesAlgebra m = (m->m->m,m)

foldParentheses ::
  ParenthesesAlgebra m -> Parentheses -> m
foldParentheses (match,empty) = fold where
  fold (Match l r) = match (fold l)
                           (fold r)
  fold Empty       = empty
```

For example, the fold can be used to compute the depth (by means of function depthParentheses) and the width (by means of function widthParentheses) of matching parenthesis in a compositional way. The depth of a string of matching parentheses s is the largest number of unmatched parentheses that occurs in a substring of s. For example, the depth of the string ((()))() is 3. The width of a string of matching parentheses s is the the number of substrings that are matching parentheses themselves, which are not a substring of a surrounding string of matching parentheses. For example, the width of the string ((()))() is 2.

```
depthParenthesesAlgebra ::
  ParenthesesAlgebra Int
depthParenthesesAlgebra =
  (\x y -> max (1+x) y,0)

widthParenthesesAlgebra  ::
  ParenthesesAlgebra Int
widthParenthesesAlgebra  =
  (\_ y -> 1+y,0)

depthParentheses  :: Parentheses -> Int
depthParentheses  =
  foldParentheses depthParenthesesAlgebra
widthParentheses  :: Parentheses -> Int
widthParentheses  =
  foldParentheses widthParenthesesAlgebra
```

Another example, which also exemplifies the use of higher-order functions, is the definition and use of a fold for expression trees with variables. The abstract syntax of expression trees is given by the following datatype:

```
data Expr = Expr `Add` Expr
          | Expr `Sub` Expr
          | Expr `Mul` Expr
          | Expr `Dvd` Expr
          | Num Value
          | Var Name
```

where Value is a (data)type for values of some kind, and Name a (data)type for names (identifiers). The fold on this datatype is defined by:

```
type ExprAlgebra a = (a->a->a  -- num
                     ,a->a->a  -- sub
                     ,a->a->a  -- mul
                     ,a->a->a  -- dvd
                     ,Value->a -- num
                     ,Name->a) -- var

foldExpr :: ExprAlgebra a -> Expr -> a
foldExpr (add,sub,mul,dvd,num,var) =
  fold where fold (e1 `Add` e2) =
               fold e1 `add` fold e2
             fold (e1 `Sub` e2) =
               fold e1 `sub` fold e2
             fold (e1 `Mul` e2) =
               fold e1 `mul` fold e2
             fold (e1 `Dvd` e2) =
               fold e1 `dvd` fold e2
             fold (Num n)       =
               num n
             fold (Var x)       =
               var x
```

To evaluate an expression, we need an environment that associates names to values. Suppose we have a type Env for environments, and an operator (?) that looks up the value of a name in an environment. An expression may be evaluated by means of the following function:

```
evalExpr     ::
  Env Name Value -> Expr -> Value
evalExpr env =
  foldExpr ((+),(-),(*),(/),id,(env?))
```

However, if we now extend the expression language with a construct for local definitions, we cannot use such an evaluation function anymore, because we cannot change the environment in the fold. Instead, we let evalExpr return a function, which, when given an environment, evaluates the expression. So a better definition of evalExpr is:

```
evalExpr  ::
  Expr -> Env Name Value -> Value
evalExpr  =
  foldExpr (<+>,<->,<*>,</>,const,flip (?))

(<+>),(<->),(<*>),(</>) ::
  (Env Name Value -> Float) ->
  (Env Name Value -> Float) ->
  (Env Name Value -> Float)
f <+> g = \env -> f env + g env
f <-> g = \env -> f env - g env
f <*> g = \env -> f env * g env
f </> g = \env -> f env / g env
```

Thus the environment becomes a 'local' value, that is, different subtrees may have different environments.

A more advanced and theoretical introduction to generic programming is given in [1].

4.3 Deforestation

In the chapter on computing with parsers we give several ways to compute with a result of a parser (an abstract syntax tree).

- Inserting semantic functions in the parser
- Applying folds to the abstract syntax
- Using classes instead of abstract syntax
- Passing an algebra to the parser

As an example we give a parser that parses sequences of matching parentheses, and a function that calculates the maximum nesting depth.

```
open   =  symbol '('
close  =  symbol ')'

parens  :: Parser Char Parentheses
parens  =  (\a b c d -> Match b d) <$>
                open
           <*> parens
           <*> close
           <*> parens
       <|> succeed Empty
```

The first version of a function that calculates the maximum nesting depth is obtained by inserting semantic functions in the parser.

```
nesting  :: Parser Char Int
nesting  =  (\a b c d -> max (1+b) d) <$>
                open
            <*> nesting
            <*> close
            <*> nesting
        <|> succeed 0
```

A second version of the nesting function is obtained by applying a fold to the abstract syntax tree returned by the parser.

```
nesting'  :: Parser Char Int
nesting'  =
  foldParentheses (\l r -> max (1+l) r,0)
  <$> parens
```

`nesting` is the deforested, and hence more efficient, version of `nesting'`, which is easier to write because it reuses the parser and the fold.

Classes can be used to implement the deforested or fused computation of a fold with a parser. This gives the desired efficient solution. For example, for the language of parentheses, we define the following class:

```
class  Parens a  where
  match  ::  a -> a -> a
  empty  ::  a
```

Note that types of the functions in the class `Parens` correspond exactly to the two types that occur in the type `ParenthesesAlgebra`. This class is used in a parser for parentheses:

```
parens'  :: Parens a => Parser Char a
parens'  =  (\a b c d -> match b d) <$>
                open
            <*> parens'
            <*> close
            <*> parens'
        <|> succeed empty
```

The algebra is implicit in this function: the only thing we know is that there exist functions `empty` and `match` of the correct type; we know nothing about their implementation. To obtain a function `parens'` that returns a value of type `Parentheses` we construct the following instance of class `Parens`.

```
instance  Parens Parentheses  where
  match  =  Match
  empty  =  Empty
```

Now we can write:

```
?(parens'::Parser Char Parentheses) "()()"
[(Match Empty (Match Empty Empty), "")
,(Match Empty Empty, "()")
,(Empty, "()()")
]
```

Note that we have to supply the type of `parens'` in this expression, otherwise Hugs doesn't know which instance of `Parens` to use. This is how we turn the implicit 'class' algebra into an explicit 'instance' algebra. Another instance of `Parens` can be used to compute the nesting depth of parentheses:

```
instance  Parens Int  where
  match b d  =  max (1+b) d
  empty      =  0
```

And now we can write:

```
?(parens'::Parser Char Int) "()()"
[(1, ""), (1, "()"), (0, "()()")]
```

So the answer depends on the type we want the function `parens'` to have. This also immediately shows a problem of this, otherwise elegant, approach: it does not work if we want to compute two different results of the same type, because Haskell doesn't allow you to define two (or more) instances with the same type. So once we have defined the instance `Parens Int` as above, we cannot use the function `parens'` to compute, for example, the width (also an `Int`) of a string of parentheses.

Using classes we implement a parser with an implicit algebra. Since this approach fails when we want to define different parsers with the same result type, we make the algebras explicit. Thus we obtain the following definition of `parens'`:

```
parens' ::
  ParenthesesAlgebra a -> Parser Char a
parens' (match,empty)  =  par'
  where
    par'  =  (\a b c d -> match b d) <$>
                 open
             <*> par'
             <*> close
             <*> par'
         <|> succeed empty
```

Note that it is now easy to define different parsers with the same result type:

```
depthParentheses  :: Parser Char Int
depthparentheses  =
  parens' (\b d -> max (1+b) d,0)
widthParentheses  :: Parser Char Int
widthparentheses  =
  parens' (\b d -> d+1,0)
```

4.4 Abstract interpretation

In the chapter on LL(1) parsing we introduce several kinds of grammar analyses, such as for determining whether or not a non-terminal can derive the empty string, and determining the set of symbols that can appear as the first symbol in a derivation from a nonterminal. These programs are examples of programs that interpret grammars abstractly. More extensive (and complicated) treatments of grammar analysis can be found in [3, 4].

5 Conclusions and future work

We have developed a course on grammars and parsing in which we introduce several advanced abstract programming concepts. Each of these concepts is illustrated or explained with Haskell code. Higher-order functions play a crucial role in the course, and it would have been more difficult to explain the advanced programming concepts using a conventional imperative or object-oriented language.

The lecture notes on grammars and parsing have reached a more or less stable state, and can be obtained via the authors or the web page of the course. We are still working on the lecture notes on implementation of programming languages, and hope to have a first version ready soon.

References

[1] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic Programming - An Introduction -. In S.Doaitse Swierstra, Pedro R. Henriques and Jose N. Oliveira, editors, *Advanced Functional Programming*, LNCS 1608, pages 28–115, Springer-Verlag, 1999.

[2] Jeroen Fokker. Functional Parsers. In J. Jeuring and E. Meijer, eds, *Advanced functional programming*, pages 1–24. LNCS 925, Springer-Verlag, 1995.

[3] J. Jeuring and S.D. Swierstra. Bottom-up grammar analysis -a functional formulation-. In Donald Sannella, editor, *Proceedings Programming Languages and Systems-ESOP '94*, pages 317–332. Springer-Verlag, LNCS 788, 1994.

[4] J. Jeuring and S.D. Swierstra. Constructing functional programs for grammar analysis problems. In *Conference Record of FPCA '95, SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture*, pages 259–269, 1995.