Constructing functional programs for grammar analysis problems

Johan Jeuring
Chalmers University of Technology and University of Göteborg
S-412 96 Göteborg, Sweden
email: johanj@cs.chalmers.se

Doaitse Swierstra
Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
email: doaitse@cs.ruu.nl

Abstract

This paper discusses the derivation of functional programs for grammar analysis problems, such as the EMPTY problem and the REACHABLE problem. Grammar analysis problems can be divided into two classes: top-down problems such as FOLLOW and REACHABLE, which are described in terms of the contexts of nonterminals, and bottom-up problems such as EMPTY and FIRST, which do not refer to contexts. In a previous paper we derive a program for bottom-up grammar analysis problems. In this paper we derive a program for top-down grammar analysis problems by transforming the specification of an arbitrary top-down problem into a program. The existence of a solution is guaranteed provided some natural conditions are satisfied. Furthermore, we describe a general transformation that applies to both classes of grammar analysis problems. The result of this transformation is a program that avoids unnecessary computations in the computation of a fixed point. Constructor classes, which are used to abstract from the notions bottom-up and top-down, are an essential ingredient of the latter derivation.

1 Introduction

Grammar analysis is performed in many different situations: Yacc tests whether or not its input grammar is LALR(1), parser generators contain functions for determining whether or not a nonterminal can derive the empty string (EMPTY) as part of determining the set of all symbols that can appear as the first symbol of a derived string (FIRST), and for determining the set of symbols that can appear as the first symbol following a string derived by a given nonterminal (FOLLOW). Other, similar, problems arise when analysing attribute dependencies in attribute grammars: determine the inherited attributes upon which a synthesised attribute depends (IS), and, conversely, determine the synthesised attributes upon which an inherited attribute depends (SI). Such problems are called *grammar analysis problems*. More examples of grammar analysis problems can be found in [16] and [18].

Grammar analysis problems can be divided into two classes: *bottom-up* and *top-down*. The difference between these classes is that the required information for a nonterminal in a top-down problem depends on the possible contexts of that nonterminal in a derivation from the start-symbol, whereas in a bottom-up problem the property we are interested in depends on the parse tree hanging under the nonterminal instance, and the contexts of the nonterminal can be ignored. Often the output of a bottom-up problem is used in a top-down problem. The specification of a grammar analysis problem determines the class to which it belongs: EMPTY, FIRST, and IS are bottom-up grammar analysis problems, the FOLLOW and SI problems belong to the top-down class.

Grammar analysis problems are described by sets of mutually recursive equations, and the solution is a fixed point of this equational system. Möncke and Wilhelm [16] observe this, and give several solutions, depending on the conditions that are satisfied, for such problems. One of the goals of this paper is to *derive* the solutions given by Möncke and Wilhelm.

In a previous paper [7] we study bottom-up grammar analysis problems. We derive a function of which the fixed point gives the solution of a bottom-up grammar analysis problem. This function is obtained by applying correctness preserving transformations to components of the expression occurring in the specification of the problem. The laws we apply are familiar laws for list-comprehensions (monads) [21], maps, and foldrs [1, 12]. Sufficient conditions for guaranteeing the existence of a fixed point emerge as a byproduct of this derivation. An important advantage of a derivation of a program is that it is clear why and where conditions are imposed upon the components of the program.

In this paper we study top-down grammar analysis problems. We derive a function of which the fixed point gives the solution of a top-down grammar analysis problem. The derivation is similar to the derivation for bottom-up grammar analysis problems, be it much simpler. The solution obtained corresponds to the iterative techniques in program flow analysis [17], and can be traced back among others to Kildall [11].

Furthermore, we apply a general transformation to the resulting fixed point solutions for bottom-up and top-down grammar analysis problems. In a first, naive, formulation of the fixed point computation each step consists of two parts: the first part moves the information from the old approximation to the right positions, guided by the productions of the grammar, and the second part does the actual computation of the new approximation. The first step can be done once, before the iteration, and thus the fixed point

is computed much faster. This transformation is a general technique which applies to all computations of fixed points where the input has to be arranged in order to compute the new information. This transformation may be compared to the transformation in which a constant expression (in our case a constant computation) is moved out of the body of a loop. Two essential ingredients of this transformation are constructor classes [9], and anamorphisms [13].

Since we use constructor classes, we have used Gofer [10] to implement the functions we have derived. Incorporating the functions for solving grammar analysis problems in parser generators such as a functional version of Yacc [20], Ratatosk [15], and Happy [5] would reduce the amount of code used in these parser generators. The complete code constructed in this paper is available by `ftp` from `ftp.cs.chalmers.se`. The code can be found in the file `pub/users/johanj/ga.gs`.

This paper is organised as follows. Section 2 defines the datatypes and functions that are used in manipulating grammars in Gofer. Section 3 introduces some of the results used in the calculation of fixed points. Section 4 defines both top-down and bottom-up grammar analysis problems, and gives some examples. Section 5 derives a program that can be used to solve top-down grammar analysis problems. Section 6 briefly reviews the solution to bottom-up grammar analysis problems as described in [7]. Section 7 applies the transformation that avoids rearranging the information to the programs for solving bottom-up and top-down grammar analysis problems. Section 8 concludes the paper.

2 Datatypes and functions for grammars in Gofer

This section defines various functions and datatypes in Gofer which are used in analysing and representing grammars.

2.1 Laws for functions on lists

The datatype *list* is a prominent datatype in the subsequent sections, and we will use a number of properties that are satisfied by strict functions defined on the datatype *list*. Map-distributivity says that the composition of two maps is a map again, i.e., for all functions `f` and `g`:

```
map f . map g  =  map (f . g)          (1)
```

Furthermore, the result of mapping the identity function over an argument is the argument itself, so `map id = id`. These equalities say that `map` is a *functor*. Function `foldr` can be distributed over `++` in the following way:

```
foldr f e (x++y)  =  foldr f (foldr f e y) x   (2)
```

Furthermore, for associative function `f` with unit `e`, we have

```
foldr f e (x++y)  =  f (foldr f e x) (foldr f e y)
```

A function can be pushed through a foldr, obtaining an extra occurrence of map, if the following conditions hold. If `h` distributes over `f`, i.e., `h (f x y) = f (h x) (h y)`, and `h e = e`, then

```
h . foldr f e  =  foldr f e . map h      (3)
```

Proofs of these equalities can be found in Bird and Wadler [3], or in the Bird-Meertens calculus [1, 12].

An important functional programming construct we use is *list-comprehension*. For example,

```
? [(x,y) | x <- [1,2], y <- [3,4]]
[(1,3),(1,4),(2,3),(2,4)]
```

We will use the following laws for list-comprehensions [21] in some calculations.

```
[t | t <- ts]  =  ts                        (4)
[f t | q]      =  map f [t | q]             (5)
[t | p,q]      =  concat [[t | q] | p]      (6)
```

where function `concat :: [[a]] -> [a]` concats a list of lists.

2.2 Terminals and nonterminals

The class `Symbol` has two functions `isT` and `isN`, which determine whether a symbol is a terminal or a nonterminal, respectively.

```
class  Symbol s  where
  isN  ::  s -> Bool
  isT  ::  s -> Bool
  isT  =   not . isN
```

For example, the type of characters can be defined as an instance of `Symbol` by

```
instance  Symbol Char  where
  isN c  =  'A' <= c  &&  c <= 'Z'
```

2.3 Grammars

A *context-free grammar* consists of sets of nonterminals, terminals, productions, and a start-symbol. In Gofer, we combine the sets of terminals and nonterminals into a set of symbols on which the functions `isN` and `isT` are defined. The type of symbols is a parameter of the definition of a context-free grammar. We represent a context-free grammar in Gofer by a pair, the first component of which denotes the start-symbol, and the second component of which denotes the productions of the grammar. The start-symbol is a symbol, and the productions of a grammar are a set of pairs the left-component of which is a symbol, and the right component of which is a list of symbols. A context-free grammar is a value of the type `Grammar s`, which is defined by

```
type Grammar s  =  (s, Table s [s])

type Table a b  =  [(a,b)]

(-!-)    :: Eq a => Table a b -> a -> b
t -!- v  =  head [b | (a,b) <- t, a == v]

(-:=-)    :: Eq a =>
             a -> b -> Table a b -> Table a b
i -:=- v  =  \t -> [ (a,if a==i then v else b)
                   | (a,b) <- t
                   ]

dom    :: Eq a => Table a b -> [a]
dom t  =  nub [a | (a,b) <- t]
```

Consider the grammar with the following productions.

$$\begin{array}{lcl} S & \rightarrow & Aa \mid Sb \\ A & \rightarrow & [\,] \mid aBc \\ B & \rightarrow & S \end{array}$$

where $[\,]$ denotes the empty string. This grammar is encoded as a value `ex` of type `Grammar Char` as follows.

```
ex  =  ('S',[('S',['A','a'])
            ,('S',['S','b'])
            ,('A',[])
            ,('A',['a','B','c'])
            ,('B',['S'])
            ]
       )
```

Function rhss takes a grammar and a nonterminal nt and returns the right-hand sides of the productions of nt. It is defined by

```
rhss       :: Eq s => Grammar s -> s -> [[s]]
rhss g nt  =  [rhs | (z,rhs) <- snd g, z == nt]
```

For example, rhss ex 'A' = [[],['a','B','c']]. Function nts takes a grammar, and returns the list of nonterminals of the grammar. We assume that for each nonterminal there exists at least one production. Let function nub remove duplicates from a list, then function nts is defined by

```
nts    :: Eq s => Grammar s -> [s]
nts g  =  dom (snd g)
```

For example, nts ex = SAB.

2.4 Contexts

A naive way to determine the terminals that can follow a nonterminal in a derivation, is to generate all the *contexts* of a nonterminal. A context of a nonterminal is a path from the start-symbol to the nonterminal, representing a derivation starting with the start-symbol. This path is a sequence of right-hand sides of productions together with an indication which of the nonterminals will be rewritten. Each element of this path is represented as a triple: the part of the right-hand side to the left of the nonterminal that will be rewritten, the nonterminal that will be rewritten, and the part of the right-hand side to the right of the nonterminal that will be rewritten. The concatenation of these three values is a right-hand side of a production of the grammar. For example, one of the contexts of nonterminal 'B' from grammar ex is the following list.

```
[(['a'],'B',['c'])
,([],'A',['a'])
,([],'S',[])
]
```

Function contexts takes a grammar g and a nonterminal nt, and returns the list of all contexts ending in nt. This function is specified in set notation as follows.

$$
\begin{aligned}
\mathit{contexts}\ g\ s &= \{([\,],s,[\,])\} \\
\mathit{contexts}\ g\ nt &= \{\ (l,nt,r) \mid x \leftarrow \mathit{contexts}\ n\ nt' \\
&\qquad\qquad ,\ l +\!\!+ [nt''] +\!\!+ r \leftarrow \mathit{rhss}\ nt' \\
&\qquad\qquad ,\ nt == nt''\}
\end{aligned}
$$

Here, s is the start-symbol of the grammar. The definition as a functional program of contexts uses a function cs, which given a grammar g, an integer n, and a nonterminal nt, returns the list of all contexts of length at most n+1 ending in nt.

```
contexts :: (Symbol s,Eq s) =>
            Grammar s -> s -> [[([s],s,[s])]]
contexts g nt = cs g infty nt

infty :: Int
infty = 1+infty

cs :: (Symbol s,Eq s) =>
      Grammar s -> Int -> s -> [[([s],s,[s])]]
cs g 0 nt  =  [[([],fst g, [])]], nt == fst g
           =  [],                 otherwise
cs g (n+1) nt  =
  cs g n nt ++ ncs
  where
  ncs  =  [ (l,nt,r):xs
          | ((l,nt,r),nt') <- ancs g nt
          , xs <- cs g n nt'
          ]

ancs :: (Symbol s,Eq s) =>
        Grammar s -> s -> [(([s],s,[s]),s)]
ancs g nt  =  [ ((l,nt,r),nt')
              | nt' <- nts g
              , rhs <- rhss g nt'
              , (l,nt'',r) <- splitr rhs
              , nt'' == nt
              ]

splitr :: Symbol s => [s] -> [([s],s,[s])]
splitr []      =  []
splitr (x:xs)  =
  map (\(l,n,r) -> (x:l,n,r)) (splitr xs)
  ++ if isN x then [([],x,xs)] else []
```

Using laws for list-comprehensions, ncs, which appears in the left-hand side expression for cs g (n+1) nt, can be rewritten as follows. Abbreviate the first qualifier in the list-comprehension for ncs by q, (l,nt,r) by lnr, and cs g n nt' by cnn.

```
    ncs
=      definition of ncs; abbreviations above
    [lnr:xs | q, xs <- cnn]
=      law (6) for list-comprehensions
    concat [[lnr:xs | xs <- cnn] | q]
=      law (5) for list-comprehensions
    concat
    [map (lnr:) [xs | xs <- cnn] | q]
=      law (4) for list-comprehensions
    concat [map (lnr:) cnn | q])
```

2.5 Parse trees

A naive way to determine whether or not the empty string can be derived from a nonterminal (the EMPTY problem), is to examine all sentences derivable from the given nonterminal. A derivation using productions of a context-free grammar corresponds with a *parse tree* or *derivation tree*, i.e., an element of the datatype Rose s, where the datatype Rose is defined by

```
data Rose a  =  Node a [Rose a]
```

All sentences derivable from a nonterminal can be obtained from all parse trees with the nonterminal in the top. Function generate of type

```
generate :: (Symbol s,Eq s) =>
            Grammar s -> s -> [Rose s]
```

generates all parse trees with a given nonterminal in the top. Note that there may be infinitely many parse trees with a given nonterminal in the top. Function `generate` generates parse trees in increasing order of height, and is defined in terms of `infty`, in a similar fashion as function `contexts`. The definition of `generate` is omitted. Function `sentence` takes a rose tree, and returns the sentence of which the rose tree is a derivation. Function `sentence` is defined by

```
sentence :: Symbol s => Rose s -> [s]
sentence (Node a x)  =
  if isT a
  then [a]
  else concat (map sentence x)
```

3 Lattices and CPOs

In Section 4 we will specify grammar analysis problems in terms of the functions `contexts` and `generate`. The specifications are nonterminating functions because of the occurrence of `infty` in the definitions of `contexts` and `generate`. To obtain terminating grammar analysis functions we will apply the *Fixed Point Fusion Theorem* in Section 5. This section introduces the fixed point fusion theorem and other necessary machinery.

3.1 Lattices

A *partial order* on a set `a` is a reflexive, antisymmetrical, and transitive binary relation on `a`. A *partially ordered set* or *poset* is a pair $(a, \leq)$ consisting of a set `a` together with a partial order $\leq$ on `a`. If it exists, `bottom` is the least element of a poset. Given elements `x`, `y` from `a`, `x` ‘`join`‘ `y`, is the least element in `a` that is greater than both `x` and `y`. Note that the join of two elements is uniquely defined when it exists. Function `lub` returns the *least upperbound* of a subset `b` of `a`.

```
lub  =  foldr join bottom
```

Function `lub` need not be defined for every subset `b` of `a`. Let $(a, \leq)$ be a poset. If for all elements `x` and `y` their join `x` ‘`join`‘ `y` exists, then $(a, \leq)$ is called a *join semilattice*. Since we assume `join` is associative, and `bottom` is the unit of `join`, function `lub` satisfies

```
lub (x ++ y)  =  lub x `join` lub y
```

In Gofer we define semilattices by means of a class.

```
class  Semilattice a  where
  join    ::  a -> a -> a
  bottom  ::  a

instance  Semilattice Bool  where
  join    =  (||)
  bottom  =  False

instance (Eq a, Ord a) => Semilattice [a] where
  join    =  \ a b -> sort (nub (a ++ b))
  bottom  =  []
```

Provided `a` is a semilattice, a third instance of the class `Semilattice` is the datatype `Lift a`, where `Lift a` is defined as follows.

```
data Lift a  =  U a | D

instance  Semilattice a => Semilattice (S a)
  where
  join = \x y -> case x of
                   D   -> y
                   U a -> case y of
                            D   -> U a
                            U b -> U (joinf a b)
  bottom = D
```

The types `[a]` and `Lift [a]` give two possibilities to implement sets as a semilattice. The difference between these types is that `[a]` has the empty set as bottom, whereas `Lift [a]` has a bottom below the empty set. Jones [8] gives a more extensive introduction to computing with lattices.

3.2 CPOs

Let `b` be a subset of a poset. `b` is said to be *directed* if every finite subset of `b` has a lub. A poset `a` is a *complete partial order* or *CPO* if it contains a bottom element, and if each directed subset of `a` has a `lub`. An element `x` of `a` is a *fixed point* of function `f :: a -> a` if `f x == x`. It is a *least fixed point* if for any other fixed point `y` of `f` we have `x` $\leq$ `y`. A function `f :: a -> b` is *monotonic* if it respects the ordering on `a`, i.e., `x` $\leq$ `y` implies `f x` $\leq$ `f y`. A function `f :: a -> b` is *continuous* if it respects `lub`s of directed subsets, i.e., if `b` $\subseteq$ `a` is a directed subset, then `f (lub b)` = `lub (map f b)`.

Let $(a, \leq)$ be a CPO with bottom $\perp$, and `g :: a -> a` a continuous function. It follows from the CPO Fixed Point Theorem I [4] that function `g` has a least fixed point μg, defined by μ`g = lub [`$g^n\perp$ `| n <- [0..]]`. The *Fixed Point Fusion* Theorem (or Plotkin's Lemma) is used to reason about fixed points. This theorem reads as follows.

$$f \perp = \perp \ \wedge \ f \cdot h = g \cdot f \quad \Rightarrow \quad f\ \mu h = \mu g$$

We use the Fixed Point Fusion Theorem and the CPO Fixed Point Theorem I as follows. Consider the function `(+1)`. Define `infty` = μ`(+1)`. Taking `h = (+1)` and writing `0` for the bottom $\perp$ of natural numbers, we get, applying the Fixed Point Fusion Theorem,

$$\texttt{f 0} = \perp \ \wedge \ \texttt{f (n+1)} = \texttt{g (f n)} \quad \Rightarrow \quad \texttt{f infty} = \mu\texttt{g}$$

Other applications of a calculus of extreme fixed points can be found in [14] and [19]

If `c` is a semilattice, and function `g :: c -> c` is monotonic, then μ`g` exists, and μ`g` = `lfp g bottom`, where function `lfp` is defined by

```
lfp f x  =  x,            f x == x
         =  lfp f (f x), otherwise
```

We have the following equality for `lfp f x`.

```
lfp f x  =  firstequal xs                    (7)
            where xs = x:map f xs
```

where function `firstequal` returns the first element that occurs twice in a row in a list.

4 Grammar analysis problems

Although in some grammar analysis problems only a property of the start-symbol of the grammar is sought, we define a grammar analysis problem to be a problem which requires finding information about all nonterminals of the grammar. This section formally defines grammar analysis problems. The first subsection gives some examples of grammar analysis problems. The second subsection defines grammar analysis problems.

4.1 Examples of grammar analysis problems

Part of determining whether or not a grammar is LL(1) consists of solving the grammar analysis problems EMPTY, FIRST, and FOLLOW. We also define the REACHABLE problem.

EMPTY

Given a grammar `g` and a nonterminal `nt` from `g`, the expression `empty g nt` is a boolean expressing whether or not it is possible to derive the empty string from `nt`, using the productions from `g`. Conventionally, if `=*=>` is the usual derivation relation using productions from grammar `g`, then

```
empty g nt  =  nt =*=> []
```

Note that the argument `g` is implicitly present in `=*=>` in the right-hand side expression. Using function `generate` instead of the derivation relation, `empty g nt` is defined as a functional program by

```
empty g nt  =
  [] /= [ xs
        | xs <- generate g nt
        , sentence xs == []
        ]
```

Note that evaluating the expression `empty g nt` may result in a nonterminating computation.

FIRST

Given a grammar `g` and a nonterminal `nt` from `g`, the expression `first g nt` is the set of terminals that can appear as the first element of a string of terminals derivable from `nt`. Conventionally, function `first` is specified by

```
first g nt  =  [ a | nt =*=> a:x, isT a]
```

Again using function `generate`, it is defined as a functional program by

```
first g nt  =
  nub [ head (sentence xs)
      | xs <- generate g nt
      , sentence xs /= []
      ]
```

REACHABLE

Given a grammar `g`, `reachable g nt` is a boolean expressing whether or not it is possible to reach `nt` from the start-symbol. Conventionally, function `reachable` is specified by

```
reachable g nt  =  S =*=> x ++ [nt] ++ y
```

where `S` is the start-symbol from `g`. In the definition as a functional program of function `reachable` we use function `contexts` instead of the derivation relation.

```
reachable g nt  =
  [] /= [xs | xs <- contexts g nt]
```

Applying equality (4) we obtain that `reachable g nt` equals `[] /= contexts g nt`.

FOLLOW

Given a grammar `g` and a nonterminal `nt` from `g`, the expression `follow g nt` is the set of terminals that can follow on `nt` in a derivation starting with the start-symbol `S` from `g`. Function `follow` is conventionally specified by

```
follow g nt = [ a | S =*=> x++[nt,a]++y, isT a]
```

The specification as a functional program of function `follow` uses a function `rc`, which takes a context of a nonterminal `nt`, and returns the symbols to the right of `nt` in this specific context. These functions are defined by

```
follow g nt  =
  nub [ foldr h [] (rc xs)
      | xs <- contexts g nt
      ]
  where
  rc = foldr (\(l,nt,r) xs -> r ++ xs) []
  h s x = [s],            isT s
        = first g s ++ x, empty g s
        = first g s,      otherwise
```

Bottom-up versus top-down

The definitions in the first two examples given above require finding information about a nonterminal, and do not refer to the context in which such a nonterminal appears. These two examples are bottom-up grammar analysis problems. The definitions in the last two examples explicitly refer to the context in which the nonterminal appears. These examples are top-down grammar analysis problems.

4.2 Grammar analysis problems

We formalise the notion of a grammar analysis problem. As explained above, there exist two kinds of grammar analysis problems.

For the EMPTY problem it is required to determine for all nonterminals `nt` from a grammar `g` whether or not it is possible to derive the empty string from nonterminal `nt`. A non-executable specification for this problem reads as follows. Given a nonterminal `nt` apply a property function `p` to each derivation tree with `nt` in the root. Function `p` determines whether or not the string represented by the derivation tree is empty,

```
p x  =  sentence x == []
```

Note that function `p` corresponds with the guard occurring in the list-comprehension in the definition as a functional program of `empty g nt`. To determine whether or not it is possible to derive the empty string from nonterminal `nt`, combine the list of results obtained by applying function `p` to all derivation trees with `nt` in the root. Function `combine` corresponds to the function `([] /=)`; the expression in front of the list-comprehension in the definition of `empty g nt`.

```
combine  =  foldr (||) False
```

`combine` equals the `lub` on the semilattice `Bool`.

For the FOLLOW problem it is required to determine for all nonterminals `nt` from a grammar `g` the set of terminals that can follow on `nt` in a derivation starting with the start-symbol from `g`. A non-executable specification for this problem reads as follows. Given a nonterminal `nt` apply a property function `p` to each context of `nt`. Function `p` determines the terminals that can follow upon `nt` in a derivation that starts with the derivation represented by the context.

```
p  =  foldr h [] . rc
  where
  rc = foldr (\(l,nt,r) xs -> r ++ xs) []
  h s x = [s],                isT s
        = first g s `join` x, empty g s
        = first g s,          otherwise
```

where `join` is the join of the semilattice `[a]`. To determine the set of all terminals that can follow on `nt`, apply function `combine` to the list of results returned by applying function `p` to all contexts. Function `combine` takes the union of these lists:

```
combine  =  foldr cup []
```

Again, function `combine` equals the `lub` of a semilattice, namely the semilattice `[a]`. Note that we could have used the semilattice `Lift [a]` instead of the semilattice `[a]`.

Generalising the patterns above, we now define a grammar analysis problem.

Definition 1 *A grammar analysis problem analyses a grammar* `g` *with respect to property function* `p : t a b -> c`*, where* `[t a b]` *is the result type of function* `generate` *or* `contexts`*, and* `c` *is an instance of the class* `Semilattice`*. It is an expression of the form* `analyse_td p g` *for top-down problems, and* `analyse_bu p g` *for bottom-up problems.*

```
tabulate      :: [a] -> (a -> b) -> Table a b
tabulate l f  =  l 'zip' map f l

nttab    :: Eq a =>
            Grammar s -> (s -> a) -> Table s a
nttab g  =  tabulate (nts g)

analyse_td p g = nttab g (lub.map p.contexts g)
analyse_bu p g = nttab g (lub.map p.generate g)
```

The four example problems given above are expressed as grammar analysis problems as follows.

```
empties     =  analyse_bu ((==[]) . sentence)
firsts      =  analyse_bu (take 1 . sentence)
reachables  =  analyse_td (const True)
follows g   =  analyse_td (foldr h [] . rc) g
  where
  rc = foldr (\(l,nt,r) xs -> r ++ xs) []
  h s  x = [s],                isT s
         = first g s `join` x, empty g s
         = first g s,          otherwise
```

5 Deriving a program for top-down grammar analysis

The execution of the expressions `analyse_bu p g` and `analyse_td p g` does not terminate because of the occurrence of `infty` in the definition of functions `generate` and `contexts`. This section derives an always terminating program that returns the value of `analyse_td p g`. This program is obtained by means of the theory given in Section 3.

`contexts g` is defined as `cs g infty`. Replacing the constant `infty` by a variable `n` in function `cs` results in the following equality for function `analyse_td`:

```
analyse_td p g  =  tdn infty
  where tdn n = nttab g (lub.map p.cs g n)
```

We use the CPO fixed point theorems to find the value of `tdn infty` in finite time. If there exists a semilattice `d` with a `bottom`, such that `tdn 0` = `bottom`, and such that

$$\texttt{tdn (n+1)} \;=\; \texttt{step (tdn n)} \qquad (8)$$

for a monotonic function `step :: d -> d`, then `tdn infty` equals the least fixed point of function `step`.

5.1 The semilattice

Each grammar analysis problem has a property function `p :: t a b -> c`, where `c` is a semilattice the elements of which correspond to the properties of individual symbols. We now construct a new semilattice `d` in which the properties for all the symbols are combined. We use `c` to construct the desired semilattice $(\texttt{d},\leq)$. Elements of `d` are lists of pairs, of which the first components are the nonterminals of the given grammar, and of which the second component are elements of the semilattice `c`. $\leq$ on `d` is the straightforward extension of $\leq$ on `c`. The bottom of `d` is called `bottoms_td` and is equal to `tdn 0`.

```
bottoms_td = (s -:=- p [([],s,[])])        (9)
             (nttab g (const bottom))
  where s = fst g
```

Notice that `bottom_td` depends implicitly on grammar `g`.

5.2 The derivation

We further reduce condition (8). Abbreviate `lub . map p` by `af` (for 'analyse function'). If there exists a function `stepf` such that

$$\texttt{af (cs g (n+1) nt)} \;=\; \texttt{stepf (tdn n) nt} \qquad (10)$$

then we have the following equality for `tdn (n+1)`.

$$\texttt{tdn (n+1)} \;=\; \texttt{nttab g (stepf (tdn n))}$$

Abstracting from `tdn n`, we define function `step` by

$$\texttt{step x = nttab g (stepf x)} \qquad (11)$$

Note that function `step` is monotonic if function `stepf` is monotonic in its first argument. So it remains to construct a monotonic function `stepf` such that equality (10) holds. For that purpose we manipulate the left-hand side of equation (10), heading towards an expression in terms of `tdn n`. An easy calculation shows that

```
af (cs g (n+1) nt) = af (cs g n nt) `join` af ncs
```

where `ncs` appears in the definition of `cs g (n+1) nt`. The left-hand argument of operator `` `join` `` can be expressed in terms of `tdn n`, since we have

$$\texttt{af (cs g n nt)} \;=\; \texttt{tdn n -!- nt}$$

where operator `-!-` is defined in Section 2. We proceed with the right-hand argument of `join`.

```
    af ncs
=     equality for ncs
    af (concat [map (lnr:) cnn | q])
=     af . concat = lub . map af
    lub (map af [map (lnr:) cnn | q])
=     law (5) for list-comprehensions
    lub [(af . map (lnr:)) cnn | q]
=     assume af . map (x:) = k x . af
    lub [k lnr (af cnn) | q]
```

Remember `cnn` abbreviates `cs g n nt'`. Since `af cnn` can be expressed in terms of `tdn n`:

$$\texttt{af cnn} \;=\; \texttt{tdn n -!- nt'}$$

it follows that if there exists a function `k` of type

```
k :: ([s],s,[s]) -> c -> c
```

such that `af . map (x:) = k x . af`, then function `stepf` can be defined by

```
stepf x nt  =  (x -!- nt) `join`
               lub [ k lnr (x -!- nt')
                   | (lnr,nt') <- ancs g nt
                   ]
```

Function `stepf` is monotonic in its first argument if function `k` is monotonic in its second argument. We have proved the following theorem.

Theorem 1 *If there exists a function* `k` *such that*

$$\texttt{af . map (x:)} \;=\; \texttt{k x . af} \qquad (12)$$

and `k` *is monotonic in its second argument, then*

$$\texttt{analyse\_td p g} \;=\; \texttt{lfp step bottoms\_td}$$

where function `step` *is defined in terms of function* `stepf` *in equation (11), and* `bottoms_td` *is defined in equation (9).*

Using this theorem we redefine function `analyse_td` such that it takes function `k` as an argument instead of predicate `p`. Function `analyse_td` takes three arguments: a function `k` satisfying the conditions of the above theorem, a value `v` which equals `p [([],fst g,[])]` (needed in the definition of `bottoms_td`) and a grammar `g`.

```
analyse_td ::
   (Symbol s, Semilattice c, Eq (Table s c)) =>
   (([s],s,[s]) -> c -> c) ->
   c -> Grammar s -> Table s c
analyse_td k v g  =  lfp step bottoms_td
```

5.3 Applications

The previous subsection derives a program for top-down grammar analysis problems, provided there exists a function `k` such that `af . map (x:) = k x . af`, and `k` is monotonic in its second argument. We verify these conditions for the two example top-down problems.

Reachable

The property function `p` for the Reachable problem is the function `const True`. The semilattice we are working in here is the semilattice of lists of pairs of which the first component is a symbol and the second component is a boolean. It is easy to prove that equality (12) holds if we define function `k` by `k x y = y`. Furthermore, this function is trivially monotonic in its second argument. It follows that:

```
reachables  =  analyse_td (\x y -> y) True
```

Follow

The property function `p` for the Follow problem is defined in Section 4. We have to find a function `k` such that

```
    lub . map p . map ((l,n,r):)
=
    k (l,n,r) . lub . map p
```

We calculate a definition of function `k` in two steps. We start with showing that there exists a function `k'` such that `p . ((l,n,r):) = k' r . p`, and then we show that `lub . map (k' r) = k' r . lub`. Taken together, these two equalities prove the above equality.

```
    p ((l,n,r):xs)
=     definition of p
    foldr h [] (rc ((l,n,r):xs))
=     definition of rc
    foldr h [] (r ++ rc xs)
=     foldr distributes over ++ (2)
    foldr h (foldr h [] (rc xs)) r
=     definition of p
    foldr h (p xs) r
```

It follows that if we define function `k'` by

```
k' r s  =  foldr h s r
```

then `p . ((l,n,r):) = k' r . p`. For the proof of the second equality `lub . map (k' r) = k' r . lub`, we apply equality (3). For this purpose we have to show that `k' r (x `join` y) = (k' r x) `join` (k' r y)`, and that `k' r bottom = bottom`. The former equality is proven by induction on `r`. In the induction proof we use the fact that function `h` distributes over `join`:

```
h z (x `join` y)  =  (h z x) `join` (h z y)
```

The latter equality and the above definition of function `k'` cannot be satisfied together. The problem is that we can not distinguish between the empty set and `bottom`. This subtle difference is important when there are nonterminals that cannot be reached from the start-symbol. Therefore, we use the semilattice `Lift [a]` instead of the semilattice `[a]`. Function `p` is now defined by `p = foldr h (U []) . rc`, and never returns the value `B`. Therefore, we can define `k' r D = D`, and we can apply equality (3) to obtain the desired equality. Function `k` defined by

```
k (l,n,r) D  =  D
k (l,n,r) s  =  foldr h s r
```

is monotonic in its second argument if function `h` is monotonic in its second argument, which is true. It follows that:

```
follows g  =
  analyse_td k (U []) g
  where
  k (l,n,r) D = D
  k (l,n,r) s = foldr h s r
  h s x = U [s],                  isT s
        = U (fg -!- s) `join` x, eg -!- s
        = U (fg -!- s),           otherwise
  eg = empties g
  fg = firsts g
```

6 Bottom-up grammar analysis

In [7] we derive a program for bottom-up grammar analysis problems that satisfy a number of properties. We repeat the main result of that paper.

Theorem 2 *Suppose there exists a function* `k` *such that*

$$\texttt{p (Node s xs)} \;=\; \texttt{k s (tabulate xs p)}$$
$$\texttt{k s} \;=\; \texttt{foldr f e}$$

where `f` *is monotonic in both of its arguments,* `e` *is the unit of* `f`*, and both* `fl y` *and* `fr x`*, defined by*

```
fl y  =  \x -> f x y
fr x  =  \y -> f x y
```

distribute over `join`. *Then*

```
analyse_bu p g  =  lfp step bottoms_bu
```

where `bottoms_bu` *is defined by*

```
bottoms_bu = nttab g (const bottom)
```

and function `step` *is defined in equation (11). Function* `stepf`*, which is used in the definition of function* `step`*, is defined by*

```
stepf x nt  =  (x -!- nt) `join`
               lub [k nt (j x rhs)
                   | rhs <- rhss g nt
                   ]
j x rhs  =  let f s = if isN s
                      then x -!- s
                      else k s []
            in map (\(a,b) -> (a,f b)) rhs
```

Using this theorem we redefine function `analyse_bu` such that it takes a function `k` that satisfies the conditions of the above theorem as an argument, instead of predicate `p`.

```
analyse_bu ::
  (Symbol s, Semilattice c, Eq (Table s c)) =>
  (s -> Table s c -> c) ->
  Grammar s -> Table s c
analyse_bu k g  =  lfp step bottoms_bu
```

6.1 Applications

Applying the above theorem to the EMPTY problem and the FIRST problem gives the following results.

```
empties  =
  analyse_bu (\s xs -> if isT s then False
                       else and (map snd xs))

firsts g  =  analyse_bu k g
where
k s xs = [s],            isT s
       = foldr j [] xs,  isN s
j (s,y) x = y,           isT s
          = y `join` x,  eg -!- s
          = y,           otherwise
eg = empties g
```

7 More efficient fixed point computations

The fixed point solutions to top-down grammar analysis problems and bottom-up grammar analysis problems given by Theorems 1 and 2, respectively, can both be written as the least fixed point of a function that for each nonterminal first arranges the elements of the previous approximation of the result, and then evaluates the arranged values for each nonterminal. In each iteration of function `lfp` the function `step` arranges information describing the approximations computed thus far, and then computes a new approximation based on these values. Since the arrangement involves many evaluations of the operator `(-!-)`, this is a costly part of the overall computation. However, arranging the input is completely independent of the value of the input. It is therefore desirable to 'factor out' the arranging function `arr` from function `lfp`. Thus we obtain a computation of a fixed point in which `arr` is evaluated once instead of at each next step of the computation of the fixed point. The gain in efficiency of this arranging transformation is linear, but it may be substantial. It corresponds directly to moving constant expressions out of a loop, as found in most modern optimising compilers for imperative languages. This section derives the program, using constructor classes [9] to abstract from the notions top-down and bottom-up in the calculations.

7.1 Rewriting the fixed point solutions

The fixed point solutions to top-down grammar analysis problems and bottom-up grammar analysis problems given by Theorems 1 and 2, respectively, can both be rewritten as follows. We separate the value independent and value dependent parts of the computation.

```
lfp (map (p2 eval) . arr) bottoms
```

where function `p2` is defined by

```
p2 f  =  \(a,b) -> (a,f b)
```

Given a type `t`, functions `arr` and `eval` have the following types.

```
arr :: Table s c -> Table s (c,t (Sum (s,c) s))
eval :: (c,t (Sum (s,c) s)) -> c
```

Function `arr` takes the old approximations, and arranges the necessary information for each nonterminal. For example, for bottom-up problems it returns for each nonterminal a pair, consisting of the old approximation for the nonterminal, and the list of right-hand sides of the nonterminal, in which each element is an element of the datatype `Sum (s,c) s`, i.e., each element is either a nonterminal together with its approximation, or a terminal.

```
data Sum a b  =  L a | R b

(-+-)     :: (a -> b) -> (c -> d) ->
             Sum a c -> Sum b d
f -+- g  =  \s -> case s of
                    L x  ->  L (f x)
                    R y  ->  R (g y)
```

Function `eval` takes the old approximation for a nonterminal and a structure containing the information needed to compute the new approximation, and returns the new approximation for the nonterminal.

The type `t` is the type `Bu` for bottom-up problems and the type `Td` for top-down problems. Since we want to use maps on these datatypes, we let them be instances of the constructor class `Functor`.

```
data Bu a  =  Bu [[a]]
data Td a  =  Td [(([a],a,[a]),a)]

instance  Functor Bu  where
  map f (Bu xs)  =  Bu (map (map f) xs)

instance  Functor Td  where
  map f (Td xs)  =
    let g (ys,n,zs) = (map f ys,f n,map f zs)
        h ((ys,n,zs),n') = (g (ys,n,zs),f n')
    in Td (map h xs)
```

The functions `arr` and `eval` are defined as follows for bottom-up and top-down problems, respectively. We assume these functions are defined in the context of a grammar `g` and an analysis function `k`.

```
arr_bu s = nttab g (\nt -> (s -!- nt,h nt))
  where h nt = map (s -!!-) (Bu (rhss g nt))

(-!!-) :: (Symbol s,Eq s) =>
          Table s c -> s -> Sum (s,c) s
l -!!- s = if isN s then L (s,l -!- s) else R s

eval_bu (c,Bu xs)  =  c `join` lub (map k xs)

arr_td s = nttab g (\nt -> (s -!- nt,h nt))
  where h nt = map (s -!!-) (Td (ancs g nt))

eval_td (c,Td xs)  =  c `join` lub (map k xs)
```

At the end of this section we will define function `analyse :: Functor t => ... t ...` which takes functions `arr` and `eval` as arguments. Thus constructor classes allow us to abstract from the notions bottom-up and top-down. In the remaining calculations of this section we will only use the type variable `t`.

7.2 Factoring out function `arr`

Each computation step of function `lfp` arranges its input. However, arranging the input is completely independent of the value of the input. It is therefore desirable to 'factor out' function `arr` from function `lfp`.

The first transformation we apply is introducing streams in the fixed point computation. For this purpose we use function `to` and `from`. Function `to` takes a list of pairs consisting of a nonterminal and a stream of approximations for the nonterminal, and returns a stream of lists of pairs of nonterminals and approximation values. It is a kind of transpose function. Function `from` is a left- and right-inverse of function `to`.

```
to    :: [(a,[c])] -> [[(a,c)]]
to xs =  map (p2 head) xs:to (map (p2 tail) xs)

from         :: [[(a,c)]] -> [(a,[c])]
from (x:xs)  =  x `cons` (from xs)

cons  =  zipWith (\(a,c) (b,cs) -> (a,c:cs))
```

Function `to` is a list *anamorphism*; function `from` is a list *catamorphism* [13]. For these functions we have

$$\texttt{to . from} \;=\; \texttt{id} \qquad (13)$$

$$\texttt{from . to} \;\leq\; \texttt{id} \qquad (14)$$

The proofs of these equations are by coinduction, and use properties of anamorphisms [13, 14] and zips [2, 6]. They are omitted for reasons of space. Functions `to` and `from` are introduced in the fixed point computation as follows.

```
    lfp f x
=     equality for fixed points (7)
    firstequal xs
    where xs = x:map f xs
=     to . from = id (13)
    firstequal (to fx)
    where fx = from xs
          xs = x:map f (to fx)
=     removing xs
    firstequal (to fx)
    where fx = from (x:map f (to fx))
=     definition of from
    firstequal (to fx)
    where fx = x `cons` from (map f (to fx))
```

We proceed with the subexpression `map f . to` from the right-hand argument of operator `` `cons` ``. We assume that `f` equals the composition of functions `map (p2 eval) . arr`.

```
    map f . to
=     definition of f
    map (map (p2 eval) . arr) . to
=     map-distributivity (1)
    map (map (p2 eval)) . map arr . to
=     assume equality (15) below
    map (map (p2 eval)) .
    to . map (p2 uzip . p3 tos) . arr
```

where function `p3` is defined by

```
p3 f  =  \(a,(b,c)) -> (a,(b,f c))
```

and function `uzip` is the function `uncurry zip`. The equality used in the above calculation pushes function `arr` through function `to`.

$$\texttt{map arr.to} \;=\; \texttt{to.map (p2 uzip.p3 tos).arr} \qquad (15)$$

The function `tos` is very similar to function `to`. It is an anamorphism that takes a `t`-structure containing streams to a stream of `t`-structures, where `t` may be either the type `Bu` or the type `Td`.

```
tos :: Functor t =>
       t (Sum (s,[c]) s) -> [t (Sum (s,c) s)]
tos xs = map (i head) xs:tos (map (i tail) xs)
  where i f = (p2 f) -+- id
```

We could have combined the definitions of `to` and `tos` in one definition, but the occurrence of `Sum` in the type for `tos` makes the resulting functions rather awkward.

For each instance `t` of the class `Functor` for which we want to apply the results of this section we have to prove equality (15). Again, these proofs are by coinduction, and omitted. We proceed the above calculation with the composition of functions `from . map (map (p2 eval)) . to . map (p2 uzip . p3 tos)`. In the first step we apply the following equation, which combines functions `to` and `from` with function `map eval`.

$$\begin{array}{l} \texttt{from . map (map (p2 eval)) . to} \\ \leq \\ \texttt{map (p2 (map eval))} \end{array} \qquad (16)$$

The proof of this equation, using amongst others equation (14), is omitted.

```
    from . map (map (p2 eval) . to
    . map (p2 uzip . p3 tos)
≤     equation (16)
    map (p2 (map eval)).map (p2 uzip.p3 tos)
=     map distributivity; property of p2
    map (p2 (map eval . uzip) . p3 tos)
```

This concludes the derivation. We have found that we can write `lfp f x` as follows.

```
firstequal (to fx)
where
fx = let h = p2 (map eval . uzip) . p3 tos
     in x `cons` map h (arr fx)
```

Since function `eval` is defined equally in terms of `map k` for top-down problems and bottom-up problems, we replace the argument `eval` by an argument corresponding to `map k` (replacing `eval` by function `k` itself makes it more complicated to describe the type of function `analyse` below). The resulting function `analyse` is given in the following theorem.

Theorem 3 *Both of the types* `Bu` *and* `Td` *are instances of the class* `Functor` *for which equation (15) holds. For these types we can write* `lfp (map (p2 eval) . arr) bottoms` *as* `analyse arr eval bottoms`*, where function* `analyse` *is defined by:*

```
analyse :: (Functor t, Eq [c], Semilattice c) =>
  ([(a,[c])] -> [(a,([c],t (Sum (a,[c]) b)))]) ->
  (t (Sum (a,c) b) -> [c]) ->
  Table a c -> Table a c
analyse arr eval x  =  firstequal (to fx)
  where
  fx = let g (b,bs) = b `join` lub (eval bs)
           h = p2 (map g . uzip) . p3 tos
       in x `cons` map h (arr fx)
```

Using function `analyse`, we define functions `analyse_bu` and `analyse_td` as follows.

```
analyse_bu eval g  =  analyse
                        arr_bu
                        eval
                        bottoms_bu
analyse_td eval g  =  analyse
                        arr_td
                        eval
                        bottoms_td
```

where value `bottoms_bu` is defined in Theorem 2, and value `bottom_td` in equation (9). Although the gain in efficiency compared with the programs in Sections 5 and 6 is linear, it may be substantial. For example, computing `follows` for a grammar that requires about forty iteration steps using the above definition of `analyse_td` is about twenty times faster than computing `follows` using the old definition of `analyse_td`.

8 Conclusions

Using laws for monads, maps and folds, we have derived a program for the top-down analysis of grammars. Together with the program for bottom-up grammar analysis derived in [7], this constitutes a complete description of programs for grammar analysis problems. Furthermore, we have given a derivation that transforms both programs for grammar analysis into a more efficient programs by avoiding the repeated arranging of information in the computation of the fixed point. Constructor classes allow us to apply this transformation to both programs in one go; without constructor classes we would have had to perform the same derivation twice. Anamorphisms and their properties are other essential ingredients of this transformation.

References

[1] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume F36 of *NATO ASI Series*, pages 5–42. Springer–Verlag, 1987.

[2] R.S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, volume F55 of *NATO ASI Series*, pages 151–216. Springer–Verlag, 1989.

[3] R.S. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall International, 1988.

[4] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[5] Andy Gill and Simon Marlow. Happy manual. Published on comp.lang.functional, 1993.

[6] J. Jeuring. *Theories for Algorithm Calculation*. PhD thesis, Utrecht University, 1993. Parts of the thesis appeared in the Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics.

[7] 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, 1994. LNCS 788.

[8] Mark P. Jones. Computing with lattices: An application of type classes. *J. Functional Programming*, 2(4):475–503, 1992.

[9] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proceedings of the 6th ACM Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, pages 52–61, 1993.

[10] Mark P. Jones. Release notes for Gofer 2.28. Included as part of the standard Gofer distribution, February 1993.

[11] G.A. Kildall. A unified approach to global program optimization. In *Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[12] L. Meertens. Algorithmics—towards programming as a mathematical activity. In J.W. de Bakker, M. Hazewinkel, and J.K. Lenstra, editors, *Proceedings of the CWI Symposium on Mathematics and Computer Science*, volume 1 of *CWI Monographs*, pages 289–334. North–Holland, 1986.

[13] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes, and barbed wire. In J. Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts*, pages 124–144, 1991.

[14] Erik Meijer. *Calculating compilers*. PhD thesis, Nijmegen University, 1992.

[15] Torben Mogensen. Ratatosk – a parser generator and scanner generator for Gofer. Published on comp.lang.functional, 1993.

[16] Ulrich Möncke and Reinhard Wilhelm. Grammar flow analysis. In *Attribute Grammars, Applications and Systems, SAGA '91*, pages 151–186. Springer-Verlag, New York, 1991. LNCS 545.

[17] S.S. Muchnick and N.D. Jones, editors. *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981.

[18] M.J. Nederhof. *Linguistic Parsing and Program Transformations*. PhD thesis, University of Nijmegen, 1994.

[19] Mathematics of Program Construction Group (Eindhoven Technical University). Fixed-point calculus. *Information Processing Letters*, 53(3):131–136, 1995.

[20] Simon L. Peyton Jones. Yacc in Sasl – an exercise in functional programming. *Software–Practice and Experience*, 15(8):807–820, 1985.

[21] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.