Bottom-up Grammar Analysis

- A Functional Formulation -

J. Jeuring and D. Swierstra

UU-CS-1994-01
January 1994

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Bottom-up Grammar Analysis

- A Functional Formulation -

J. Jeuring and D. Swierstra

Technical Report UU-CS-1994-01
January 1994

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0924-3275

Bottom-up Grammar Analysis — A Functional Formulation —[*]

Johan Jeuring and Doaitse Swierstra
Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
email: {johan,doaitse}@cs.ruu.nl

Abstract

This paper discusses bottom-up grammar analysis problems such as the EMPTY problem and the FIRST problem. It defines a general class of bottom-up grammar analysis problems, and from this definition it derives a functional program for performing bottom-up grammar analysis. The derivation is purely calculational, using theorems from lattice theory, the Bird-Meertens calculus, and laws for list-comprehensions. Sufficient conditions guaranteeing the existence of a solution emerge as a byproduct of the calculation. The resulting program is used to construct programs for the EMPTY problem and the FIRST problem.

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 upon 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*. 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 for that nonterminal, whereas in a bottom-up problem the contexts of a 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. This paper studies bottom-up grammar analysis.

Grammar analysis problems are described by sets of mutually recursive equations, and the solution of a grammar analysis problem is a fixed point of this equational system. Möncke and Wilhelm [9] observe this, and give several solutions, depending on the conditions that are satisfied, for such problems. The goal of this paper is to *derive* the solutions given by Möncke and Wilhelm. We start with a very general specification of a bottom-up grammar analysis problem, and we derive a function of which the fixed point gives the solution of the problem. This function is obtained by applying laws to components of the expressions occurring in the specification. The laws we apply are familiar laws for, for example, list-comprehensions [11], and maps [1, 7]. Sufficient conditions for guaranteeing the existence of a fixed point solution emerge as a byproduct of this derivation. Finally we give the implementation of the derived algorithm in the functional language Gofer [5, 3]. Incorporating the functions for solving grammar analysis problems in parser generators such as a functional version of Yacc [10] and Ratatosk [8] would reduce the amount of code used in, and very likely increase the speed of, these parser generators.

[*]This paper is an extended version of a paper with the same title that will be presented at ESOP '94.

This paper is organised as follows. Section 2 defines the datatypes that are used in manipulating grammars in Gofer. Section 3 introduces some necessary concepts of lattice theory needed in the subsequent sections. Section 4 defines the class of bottom-up grammar analysis problems, and gives some examples. Section 5 derives an algorithm that can be used to solve bottom-up problems. Section 6 concludes the paper.

2 Datatypes for grammars in Gofer

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

Functions
We use simple juxtaposition and a little white space to denote the application of a function $f : s \to t$ to an argument $x \in s$, i.e., $f\ x$. Composition of functions $f : s \to t$ and $g : r \to s$ is written $f \cdot g : r \to t$. Composition is associative, and the identity function *id* is the unit of composition. Projection *exl* (*exr*) selects the left (right) component of a pair, i.e,

$$\begin{aligned} exl\ (a,b) &= a \\ exr\ (a,b) &= b \end{aligned}$$

Given functions $f : A \to B$ and $g : A \to C$, function $f \vartriangle g : A \to B \times C$ (split) applies both f and g to an argument. The type $B \times C$ is the cartesian product of the sets B and C.

$$(f \vartriangle g)\ a = (f\ a, g\ a)$$

Given functions $f : A \to B$ and $g : C \to D$, function $f \times g : A \times C \to B \times D$ (product) applies f to the first component, and g to the second component of its argument.

$$(f \times g)\ (a,c) = (f\ a, g\ c)$$

We have the following laws concerning projections, split, and product.

$$\begin{aligned} exl \cdot (f \vartriangle g) &= f & (1)\\ exr \cdot (f \vartriangle g) &= g & (2)\\ exr \cdot f \times g &= g \cdot exr & (3)\\ exl \cdot f \times g &= f \cdot exl & (4)\\ f \times g \cdot h \vartriangle j &= (f \cdot h) \vartriangle (g \cdot j) & (5) \end{aligned}$$

Function application binds stronger than a binary operator, and among the binary operators function composition binds weakest.

Lists
The datatype *list* is a prominent datatype in the subsequent sections, and we will use a number of properties that are satisfied by functions defined on the datatype *list*. The empty list is denoted by $[\,]$, and the concatenation of two lists x and y is denoted by $x +\!\!+ y$. Prepending an element x to a list xs is denoted by $x : xs$. The datatype *list* over base type A is denoted by $A*$. For $f : A \to B$, function $f* : A* \to B*$, called a *map* function takes a list and applies function f to all elements in the list, so

$$f*\ xs = [f\ x \mid x \leftarrow xs]$$

For the map function we have

$$\begin{aligned} f*\ [\,] &= [\,] \\ f*\ (x +\!\!+ y) &= f*\ x +\!\!+ f*\ y & (6)\\ f*\ (x : xs) &= f\ x : f*\ xs \end{aligned}$$

Map-distributivity says that the composition of two maps is a map again, i.e., for all functions f and g:

$$f* \cdot g* = (f \cdot g)* \qquad (7)$$

Furthermore, the result of mapping the identity function over an argument is the argument itself, so

$$id_A* = id_{A*}$$

These equalities say that $*$ is a *functor*. An important functional programming construct we use is *list-comprehension*. For example,

$$\begin{aligned} & [(x,y) \mid x \leftarrow [1,2], y \leftarrow [3,4]] \\ = \\ & [(1,3),(1,4),(2,3),(2,4)] \end{aligned}$$

We will use the following laws for list-comprehensions [11] in some calculations.

$$\begin{aligned} [t \mid t \leftarrow ts] &= ts & (8)\\ [f\ t \mid q] &= f*\ [t \mid q] & (9)\\ [t \mid p, q] &= concat\ [[t \mid q] \mid p] & (10) \end{aligned}$$

where function *concat* flattens a list of lists. Function *concat* is defined in terms of the reduce operator. The *reduce* operator / takes an associative operator $\oplus$ with unit $1_\oplus$, and a list, and places the operator in between the elements of a list, so $\oplus/\ [a,b,c] = a \oplus b \oplus c$. For operator $\oplus : A \times A \to A$ we have $\oplus/ : A* \to A$. It can be defined by

$$\begin{aligned} \oplus/\ [\,] &= 1_\oplus \\ \oplus/\ (x +\!\!+ y) &= \oplus/\ x \oplus \oplus/\ y & (11)\\ \oplus/\ (x : xs) &= x \oplus \oplus/\ xs \end{aligned}$$

Function *concat* is an example of a reduce: it is defined by *concat* = $+\!\!+/$. For the composition of a map and function $+\!\!+/$, and for the composition of a reduce and function $+\!\!+/$ we have

$$f* \cdot +\!\!+/ \quad = \quad +\!\!+/ \cdot f** \qquad (12)$$
$$\oplus/ \cdot +\!\!+/ \quad = \quad \oplus/ \cdot (\oplus/)* \qquad (13)$$

Terminals and nonterminals

Suppose a terminal is a value of type b, and a nonterminal is a value of type a. A symbol that is either a nonterminal of type a or a terminal of type b is a value of the datatype *Symbol* defined as

$$data\ Symbol\ a\ b \quad = \quad N\ a \mid T\ b$$

An element $N\ x$ is considered to be a nonterminal, and an element $T\ y$ is considered to be a terminal. Note that this definition makes the conventional disjoint sum operational.

Grammars

A context-free grammar consists of sets of nonterminals, terminals, productions, and a start-symbol. In Gofer, the sets of nonterminals and terminals correspond with the types a and b, respectively. These types are parameters 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 nonterminal, i.e., a value of type a. The productions of a grammar are a set of pairs the left-component of which is a non-terminal, and the right component of which is a list of symbols. A context-free grammar is a value of the type *Grammar*, which is defined by

$$\begin{array}{l} \quad type\ Grammar\ a\ b \\ = \\ \quad (a, [(a, [Symbol\ a\ b])]) \end{array}$$

For example, consider the grammar *eg* for expressions in a variable v.

$$\begin{array}{lcl} E & \rightarrow & E{+}T \mid T \\ T & \rightarrow & T{*}F \mid F \\ F & \rightarrow & (E) \mid v \end{array}$$

This grammar is represented in Gofer by

$$\begin{array}{lcl} eg & = & (E, [(E, [T\ +, N\ T]) \\ & & \quad ,(E, [N\ T]) \\ & & \quad ,(T, [N\ T, T\ *, N\ F]) \\ & & \quad ,(T, [N\ F]) \\ & & \quad ,(F, [T\ (, N\ E, T\ )]) \\ & & \quad ,(F, [T\ v])] \\ & &) \end{array}$$

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

$$rhss\ g\ nt \quad = \quad [rhs \mid (nt, rhs) \leftarrow exr\ g]$$

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. Function *nts* is defined by

$$nts\ g \quad = \quad nub\ (exl*\ (exr\ g))$$

where function *nub* removes duplicates from a list.

Parse Trees

To determine whether or not the empty string can be derived from a nonterminal (the EMPTY problem), we have to refer to all sentences that are derivable from the given nonterminal in the given grammar. A derivation using productions of a context-free grammar corresponds to a *parse tree* or *derivation tree*, i.e., an element of the datatype *Rosetree*, which is defined by

$$\begin{array}{l} \quad data\ Rosetree\ a\ b \\ = \\ \quad Node\ a\ [Rosetree\ a\ b] \mid Leaf\ .b \end{array}$$

For example, the following derivation of the sentence $v{+}v$ using the productions from grammar *eg*

$$\begin{array}{l} \quad E \\ \Rightarrow \\ \quad E{+}T \\ \Rightarrow \\ \quad T{+}T \\ \Rightarrow \\ \quad F{+}T \\ \Rightarrow \\ \quad v{+}T \\ \Rightarrow \\ \quad v{+}F \\ \Rightarrow \\ \quad v{+}v \end{array}$$

corresponds to the derivation tree *dt* defined by

```
Node E
   [Node E
      [Node T
         [Node F
            [Leaf v]]],
    Leaf +,
    Node T
      [Node F
         [Leaf v]]
   ]
```

Suppose function $top : Rosetree\ a\ b \rightarrow Symbol\ a\ b$ returns the top of a rose-tree. For each subtree of a derivation tree of the form $Node\ a\ x$ we have that $a \rightarrow top{*}\ x$ is a production of the grammar.

The function *sen* takes a rose-tree, and returns the sentence of which the rose-tree is a derivation. Funcion *sen* is defined by

$$\begin{aligned} sen\ (Node\ a\ x) &= \mathbin{+\!\!+}/\ (sen{*}\ x) \\ sen\ (Leaf\ b) &= [b] \end{aligned}$$

It follows that $sen\ dt = v{+}v$.

Catamorphisms on Rose-Trees
For every recursive datatype we can define a function which recursively replaces constructors by functions [6]. By definition, a *catamorphism* on the datatype *Rosetree* is a function $h : Rosetree\ a\ b \rightarrow c$ that is uniquely determined by functions f and g as follows.

$$\begin{aligned} h\ (Node\ a\ x) &= f\ a\ (h{*}\ x) \\ h\ (Leaf\ b) &= g\ b \end{aligned}$$

For such a function h we write

$$h = RT\_cata\ f\ g$$

The function *sen* defined above is a catamorphism, i.e.,

$$\begin{aligned} sen &= RT\_cata\ s\ t \\ &\quad \textbf{where}\ s\ a\ x = \mathbin{+\!\!+}/\ x \\ &\qquad\qquad t\ b = [b] \end{aligned}$$

Another example of a catamorphism on *Rosetree* is the *height* function, which returns the height of a rose-tree.

$$\begin{aligned} height &= RT\_cata\ s\ t \\ &\quad \textbf{where}\ s\ (N\ a)\ x = 1 + \uparrow/\ x \\ &\qquad\qquad t\ (T\ b) = 1 \end{aligned}$$

where $\uparrow$ returns the maximum of two numbers. For example, $height\ dt = 5$. We will encounter several other catamorphisms on rose-trees in the following sections.

2.1 Implementation

The definitions of some of the functions and datatypes given above are translated into Gofer as follows.

```
split f g x = (f x , g x)

data Symbol a b = N a
                | T b

type Grammar a b = (a,[(a,[Symbol a b])])

rhss :: Eq a =>
        Grammar a b ->
        a ->
        [[Symbol a b]]
rhss g nt = [rhs
            | (z,rhs) <- snd g
            , z==nt
            ]

nts :: Eq a => Grammar a b -> [a]
nts g = nub (map fst (snd g))
```

3 Lattice theory

This section only gives the definitions of notions from lattice theory that are used in the subsequent sections. For a more extensive introduction to lattice theory the reader is referred to e.g. [2].

Lattices and CPO's
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 $(D, \sqsubseteq)$ consisting of a set D together with a partial order $\sqsubseteq$ on D. If it exists, the least or *bottom* element of a poset is usually denoted by $\bot$. Given $d, d' \in D$, their *join*, denoted by $d \sqcup d'$, is the least element in D that is greater than both d and d'. It is fully characterised by the following equation:

$$\begin{aligned} & c = d \sqcup d' \\ \equiv \quad & \\ & (\forall e :: c \sqsubseteq e \ \equiv\ d \sqsubseteq e \ \wedge\ d' \sqsubseteq e) \end{aligned}$$

Note that the join of two elements in D is uniquely defined when it exists. The *least upperbound* or *lub* of a subset $X \subseteq D$ is denoted by $\sqcup/\ X$. It is defined by

$$c = \sqcup/\ X$$

$$\equiv$$
$$(\forall e :: c \sqsubseteq e \;\equiv\; (\forall x : x \in X : x \sqsubseteq e))$$

Not every $X \subseteq D$ needs to have a lub. The *meet* $\sqcap$ and *greatest lowerbound* or *glb* are dual to the join and the lub, respectively. Their definitions are omitted. Let $(D, \sqsubseteq)$ be a poset. If for all elements d and d' their join $d \sqcup d'$ exists, then $(D, \sqsubseteq)$ is called a *join semilattice.* A *meet semilattice* is defined similarly. Let S be a subset of a poset. S is said to be *directed* if every finite subset of S has an upper bound. A poset D is a *complete partial order* or *cpo* if it contains a bottom element, and if each directed subset of D has a lub, so $\sqcup/\ X$ exists for all directed subsets $X \subseteq D$.

Fixed Points

An element $d \in D$ is a *fixed point* of function $f : D \to D$ if $f\ d = d$. It is a *least fixed point* if for any other fixed point d' of f we have $d \sqsubseteq d'$. A function $f : D \to E$ is *monotonic* if it respects the ordering on D, i.e.,

$$d \sqsubseteq d' \quad\Rightarrow\quad f\ d \sqsubseteq f\ d'$$

A function $f : D \to E$ is *continuous* if it respects lubs of directed subsets, i.e., if $X \subseteq D$ is a directed subset, then

$$(f \cdot \sqcup/)\ X \quad = \quad (\sqcup/ \cdot f{*})\ X$$

Let D be a finite set, $(D, \sqsubseteq)$ a CPO with bottom $\bot$, and $g : D \to D$ a continuous function. It follows from the CPO Fixed Point Theorem I [2] that function g has a least fixed point μg, defined by

$$\mu g \quad = \quad \sqcup/\ [g^n \perp \mid n \leftarrow [0..]]$$

Since $g^i \perp \sqsubseteq g^{i+1} \perp$, we have that the least fixed point of g equals the first element in $[g^n \perp \mid n \leftarrow [0..]]$ that occurs twice, i.e.,

$$\sqcup/\ [g^n \perp \mid n \leftarrow [0..]] \quad = \quad \mathit{lfp}\ g \perp$$

where function *lfp* is defined by

$$\mathit{lfp}\ f\ x \quad = \quad \begin{cases} x & \text{if } f\ x = x \\ \mathit{lfp}\ f\ (f\ x) & \text{otherwise} \end{cases}$$

The *Fixed Point Fusion* Theorem (or Plotkin's Lemma) is used to reason about fixed points. This theorem reads as follows.

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

We use the Fixed Point Fusion Theorem and the CPO Fixed Point Theorem I as follows. Consider the function $(+1)$. Define $\infty = \mu(+1)$. Taking $h = (+1)$, applying the Fixed Point Fusion Theorem gives

$$f \perp = \perp \;\wedge\; f \cdot (+1) = g \cdot f$$
$$\Rightarrow$$
$$f\ \mu(+1) = \mu g$$

Writing ∞ for $\mu(+1)$, and 0 for the bottom $\perp$ of the natural numbers, we get

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

3.1 Implementation

The definitions of some of the functions and classes given above are translated into Gofer as follows.

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

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

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

lfp :: Eq a =>
       (a -> a) ->
       a ->
       a
lfp f x | x == f x  = x
        | otherwise = lfp f (f x)

lub :: Semilattice a => [a] -> a
lub = foldl join bottom
```

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 defines bottom-up grammar analysis problems. The first subsection gives some examples of grammar analysis problems. The second subsection discusses functions for generating derivation trees. The third subsection defines bottom-up grammar analysis problems, and, finally, the fourth subsection gives the implementation of some of the functions introduced in this section.

4.1 Examples of grammar analysis problems

Part of determining whether or not a grammar is LL(1) consists of computing lookahead sets. If the grammar analysis problems EMPTY, FIRST, and FOLLOW have been solved, we can easily approximate the lookahead sets. The definitions of these problems are our first three examples. The fourth example, LEFT-CONTEXT concerns the computation of left-contexts of nonterminals. Left-contexts of nonterminals are used to determine whether or not a grammar is LR(0).

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. It is defined by

$$\begin{array}{l} \quad \mathit{Empty}\ g\ nt \\ = \\ \quad [\,] \neq [x \mid nt \overset{*}{\Rightarrow} x, x = [\,]] \end{array}$$

where $\overset{*}{\Rightarrow}$ denotes a derivation with productions from g.

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 sentence derivable from nt. It is defined by

$$\begin{array}{l} \quad \mathit{First}\ g\ nt \\ = \\ \quad \mathit{nub}\ [a \mid nt \overset{*}{\Rightarrow} [a] +\!\!+\, x, x \in X*] \end{array}$$

where X is the set of terminals of g.

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-nt S from g. It is defined by

$$\begin{array}{l} \quad \mathit{Follow}\ g\ nt \\ = \\ \quad \mathit{nub}\ [a \in X \mid S \overset{*}{\Rightarrow} u +\!\!+\, [nt, a] +\!\!+\, v] \end{array}$$

LEFT-CONTEXT
A left-context of a nonterminal is a list of terminals and nonterminals that can appear before the nonterminal in a right-most derivation from the start-nt, provided the list of symbols after the nonterminal is a list of terminals. Given a grammar g and a nonterminal nt from g, the expression *LC g nt* is the set of left-contexts of nt.

$$\begin{array}{l} \quad \mathit{LC}\ g\ nt \\ = \\ \quad \mathit{nub}\ [u \mid S \overset{*}{\underset{rm}{\Rightarrow}} u +\!\!+\, [nt] +\!\!+\, v, v \in X*] \end{array}$$

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 of the last two examples explicitly refer to the context in which the nonterminal appears, namely $u +\!\!+\, [\_, a] +\!\!+\, v$, and $u +\!\!+\, [\_] +\!\!+\, v$, respectively. These two examples are top-down grammar analysis problems. In the rest of the paper we limit ourselves to bottom-up problems.

4.2 Generating trees

The definitions in the examples of grammar analysis problems given in the previous subsection typically refer somehow to all sentences derivable from a nonterminal. The sentences derivable from a nonterminal can be obtained from the derivation trees of the grammar with the given nonterminal in the root. In this subsection we define a function returning all possible derivation trees of a grammar.

Function *generate* takes a grammar, and returns a list of lists, in which each list contains all derivation trees with the same nonterminal in the root. Before we give the definition, we discuss the function *cp* (cartesian product), which is used in the definition of function *generate*.

Function *cp*

Function *cp* returns the cartesian product of a list of lists. It is defined as a map followed by a reduce by

$$\begin{aligned} cp &= \between/ \cdot [\cdot]{**} \\ xs \between ys &= [x +\!\!+\, y \mid x \leftarrow xs, y \leftarrow ys] \end{aligned}$$

where $[\cdot]$ takes an element a, and returns the singleton list containing that element: $[a]$. Note that $[[\,]]$ is the unit of operator $\between$. Function *cp* commutes with function $f{**}$ for all functions f, i.e., for all functions f we have

$$f{**} \cdot cp \;=\; cp \cdot f{**} \qquad (14)$$

Function *generate*
Function *gh* is defined in the context of a grammar g, which from now on is considered a constant. It takes a natural number n, and a symbol nt, and returns the collection of all derivation trees, of height at most n, derivable with the productions of g with symbol nt in the root, so

$$\begin{aligned} &gh : nat \to Symbol\ a\ b \to (Rosetree\ a\ b)* \\ &gh\ n\ nt \;=\; [y \mid nt \overset{*}{\Rightarrow} y \ \wedge\ height\ y \leq n] \end{aligned}$$

where we suppose that $\overset{*}{\Rightarrow}$ derives derivation trees instead of strings with productions from grammar g. So *dt* is an element of *gh* 5 (N E). Function *generate* is defined in terms of function *gh* as follows.

$$generate\ g \;=\; (gh\ \infty \cdot N)*\ (nts\ g)$$

Function *gh* can be defined recursively in various ways; we have chosen the following definition which is easily manipulated in calculations. Function *gh* is defined by pattern matching on its first argument, using the second argument to break the tie. There are no trees of height zero, so

$$gh\ 0\ symbol \;=\; [\,]$$

There is just one derivation tree of height at most $n+1$ that can be built from a terminal.

$$gh\ (n+1)\ (T\ b) \;=\; [Leaf\ b]$$

The list of derivation trees of height at most $n+1$ derivable from a nonterminal *nt* contains the list of the derivation trees of height at most n derivable from *nt*. Furthermore, for each production for *nt* we add the cartesian product of the derivation trees of height at most n of the symbols of the right-hand side of a production for *nt*; each element of the cartesian product is turned into a derivation tree using function *Node nt*.

$$\begin{aligned} &gh\ (n+1)\ (N\ a) \\ =& \qquad\qquad\qquad\qquad (15) \\ &(gh\ n\ (N\ a)) +\!\!+ [Node\ a\ c \mid \\ &rhs \leftarrow rhss\ g\ a, c \leftarrow cp\ ((gh\ n)*\ rhs)] \end{aligned}$$

We do not bother about duplicate elements in *gh n nt*; applying function *nub* to the right-hand expression of the last equation would have removed them. The right-hand side argument of $+\!\!+$ in the last equation of the definition of function *gh* can be rewritten using laws for list-comprehensions.

$$\begin{aligned} &[Node\ nt\ c \mid \\ &rhs \leftarrow rhss\ g\ nt, c \leftarrow cp\ ((gh\ n)*\ rhs)] \\ =& \quad \text{equation (9) for list-comprehensions} \\ &(Node\ nt)* \\ &[c \mid rhs \leftarrow rhss\ g\ nt, c \leftarrow cp\ ((gh\ n)*\ rhs)] \\ =& \quad (10), (8), \text{and } (9) \\ &((Node\ nt)* \cdot +\!\!+/ \cdot cp* \cdot (gh\ n){**}) \\ &[rhs \mid rhs \leftarrow rhss\ g\ nt] \\ =& \quad (8), \text{definition of function } rhss \\ &((Node\ nt)* \cdot +\!\!+/ \cdot cp* \cdot (gh\ n){**}) \\ &(rhss\ g\ nt) \end{aligned}$$

This equality will be used in the calculation in Section 5.

4.3 Bottom-up problems

We formalise the notion of a grammar analysis problem. In case of the EMPTY problem, we want 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* we apply a 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, i.e.,

$$p \;=\; ([\,] =) \cdot sen$$

Note that function p corresponds with the two expression $nt \overset{*}{\Rightarrow} x, x = [\,]$ occurring in the list-comprehension in the definition of *Empty g nt*. To determine whether or not it is possible to derive the empty string from nonterminal *nt*, we apply the function *combine* to the list of results obtained by applying function p to all derivation trees with nt in the root. Function *combine* is defined by

$$combine \;=\; \vee/$$

Note that function *combine* corresponds with function $([\,] \neq)$ occurring in the definition of *Empty g nt*, i.e., we have

$$\begin{array}{lcl} [\,] \neq [\,] & = & false \\ ([\,] \neq)\ (x +\!\!+\, y) & = & ([\,] \neq x) \vee ([\,] \neq y) \end{array}$$

Generalising this pattern, we now define the class of bottom-up grammar analysis problems.

(16) Definition *A bottom-up grammar analysis problem, which analyses a grammar g with respect to a function p :* Rosetree *$a\ b \to c$, and an operator $\oplus : c \times c \to c$ with unit $1_{\oplus}$, is an expression of the form ag g p $\oplus$, where function ag is defined as follows.*

$$\begin{array}{ll} & ag\ g\ p\ \oplus \\ = & \\ & (id \vartriangle (af \cdot gh \ \infty \cdot N))*\ (nts\ g) \\ & \textbf{where} \\ & \begin{array}{lcl} af & = & combine \cdot properties \\ properties & = & p* \\ combine & = & \oplus/ \end{array} \end{array}$$

In case of the bottom-up grammar analysis problem EMPTY we may now write

$$empties\ g \;=\; ag\ g\ (([\,] =) \cdot sen)\ \vee$$

and in case of the bottom-up grammar analysis problem FIRST we write

$$firsts\ g \;=\; ag\ g\ (take\ 1 \cdot sen)\ \cup$$

where operator $\cup$ is defined by $x \cup y = nub\ (x +\!\!+\, y)$.

4.4 Implementation

The definitions given above are translated into Gofer as follows.

We do not give the implementation of function *ag*, because the resulting program would not terminate. In the following subsection we transform function *ag* such that it always terminates, and the resulting program can be found at the end of the following section.

We give the definitions of functions *empties* and *firsts*. There are some differences with the definitions given above. First, the last argument of the functions given above does not appear in the Gofer definitions below. We will assume later that the last argument of a bottom-up problem is always the join of a semi-lattice, so we need not pass it as an argument. The second difference is that function p is replaced by two functions, which are obtained by writing p as something very much like a catamorphism on *Rosetree*, namely, we will assume later that there exist functions *pn* and *pl* such that

$$\begin{array}{lcl} p\ (Node\ nt\ x) & = & pn\ nt\ ((top \vartriangle p)*\ x) \\ p\ (Leaf\ x) & = & pl\ x \end{array}$$

The second argument of *ag* is derived from function *pn*, and the third argument of *ag* is the function *pl*. At the end of the following section we construct these definitions.

```
empties :: Eq [(a,Bool)] =>
           Grammar a b ->
           [(a,Bool)]
empties g =
   ag
   g
   (\nt x -> and (map snd x))
   (\a -> False)

firsts :: (Eq [(a,Bool)]
          ,Eq [(a,[b])]
          ,Eq [[b]]
          ,Semilattice [b]
          ) =>
          Grammar a b ->
          [(a,[b])]
firsts g =
   ag
   g
   (\nt x -> foldr t bottom x)
   (\b -> [b])
   where
   t (N a,y) x
      | eg `at` a = nub (y ++ x)
      | otherwise = y
   t (T b,y) x = y
   eg = empties g
```

From these definitions we obtain the following type for function *ag*.

```
ag :: (Eq a
,Eq [(a,c)]
,Eq [c], Semilattice c
) =>
Grammar a b ->
(a -> [(Symbol a b,c)] -> c) ->
(b -> c) ->
[(a,c)]
```

5 The derivation of an algorithm

Function *ag* can be implemented in a functional language, but executing $ag\ g\ p\ \oplus$ will result in a nonterminating computation because of the occurrence of ∞ in the definition of function *ag*. This section derives an algorithm that can be implemented as an always terminating program that returns the value of $ag\ g\ p\ \oplus$. To obtain this algorithm we use the lattice theory given in Section 3.

Function *ag* satisfies the following equality.

$$\begin{array}{l} \quad ag\ g\ p\ \oplus \\ = \\ \quad agn\ \infty \\ \quad \textbf{where} \\ \quad agn\ n\ =\ (id \vartriangle (af \cdot gh\ n \cdot N))*\ (nts\ g) \\ \quad af\ =\ combine \cdot properties \\ \quad properties\ =\ p* \\ \quad combine\ =\ \oplus/ \end{array}$$

This expression is obtained from the definition of function *ag* in Definition (16) by replacing the constant ∞ by a variable n.

The CPO fixed point theorems may be used to find the value of $agn\ \infty$ in finite time. Suppose there exists a function K such that for $n \geq 0$

$$agn\ (n{+}1)\ =\ K\ (agn\ n) \qquad (17)$$

If we suppose furthermore that there exists a CPO $(E, \sqsubseteq_E)$ with bottom $agn\ 0$, then the results in Section 3 show that function $K : E \to E$ has a least fixed point μK, defined by

$$\mu K\ =\ \sqcup/\ [K^n\ (agn\ 0) \mid n \leftarrow [0..]]$$

provided function K is continuous, and

$$agn\ \infty\ =\ \mu K$$

The domains used in the grammar analysis problems are finite, that is, the target type E of function *ag* is a finite type. Since every finite join semilattice is a CPO, and since each monotonic function on a finite domain is continuous, it suffices to find a join semilattice with bottom $agn\ 0$, and a monotonic function K satisfying (17).

This section consists of five subsections. The first subsection constructs a join semilattice with bottom $agn\ 0$ for bottom-up grammar analysis problems. The second subsection derives a definition of function K that satisfies equation (17), i.e., it expresses $agn\ (n{+}1)$ in terms of $agn\ n$. In order to find a definition of function K that satisfies equation (17) it will be advantageous to impose conditions upon the components of the bottom-up grammar analysis problem. The third subsection shows that provided some further conditions are satisfied the function K obtained in the derivation is monotonic. The fourth subsection discusses the conditions imposed thus far, and the fifth subsection applies the derived theory to some examples.

5.1 Constructing a join semilattice with bottom *agn* 0

We want to construct a join semilattice $(E, \sqsubseteq_E)$ with bottom $agn\ 0$ and join $\sqcup_E$, such that there exists a monotonic function $K : E \to E$ satisfying $agn\ (n{+}1)\ =\ K\ (agn\ n)$. For that purpose, we impose our first condition on bottom-up grammar analysis problems.

For value $agn\ 0$ we calculate as follows.

$$\begin{array}{ll} & agn\ 0 \\ = & \quad \text{definition of } agn \\ & (id \vartriangle (af \cdot gh\ 0 \cdot N))*\ (nts\ g) \\ = & \quad \text{definition of } gh,\ a^\bullet\ b = a \text{ for all } a \\ & (id \vartriangle (af \cdot [\,]^\bullet))*\ (nts\ g) \\ = & \quad f \cdot a^\bullet = (f\ a)^\bullet \\ & (id \vartriangle (af\ [\,])^\bullet)*\ (nts\ g) \\ = & \quad \text{definition of } af \\ & (id \vartriangle 1_\oplus^\bullet)*\ (nts\ g) \end{array}$$

We have derived the following equality.

$$agn\ 0\ =\ (id \vartriangle 1_\oplus^\bullet)*\ (nts\ g)$$

i.e., $agn\ 0$ is a list of length equal to the number of nonterminals of g, of which the second components are all equal to $1_\oplus$. This suggests to construct the following join semilattice. Let E be the set of lists x of length equal to the number of nonterminals of g of which $exl{*}\ x = nts\ g$, and of which the second component of each element is an element of c, the result type of operator $\oplus$.

For the definition of the relation $\sqsubseteq_E$ and the join $\sqcup_E$, we suppose that there exists a relation $\sqsubseteq_c$ such that $(c, \sqsubseteq_c)$ with join $\sqcup_c$ is a join semilattice, and such that the unit $1_\oplus$ of operator $\oplus$ occurring in the definition of a bottom-up grammar analysis problem is the bottom of c.

Both the relation $\sqsubseteq_E$ and the join $\sqcup_E$ are now straightforward extensions of $\sqsubseteq_c$ and $\sqcup_c$, respectively. Relation $\sqsubseteq_E$ is defined by pairwise comparing elements with $\sqsubseteq_c$.

$$\begin{aligned}&x \sqsubseteq_E y\\ \equiv\ &\\ &and\ (exr{*}\ x \Upsilon_{\sqsubseteq_c} exr{*}\ y)\end{aligned}$$

where function *and* is the reduce $\wedge/$, and where $\Upsilon_\oplus$, with $\oplus$ a binary function, zips two lists of equal length to a list of pairs, and then applies operator $\oplus$ to all pairs in the list. The join of two elements is defined by pairwise joining the second components of the pairs.

$$\begin{aligned}&x \sqcup_E y\\ =\ &\\ &exl{*}\ x\ \Upsilon\ (exr{*}\ x \Upsilon_{\sqcup_c} exr{*}\ y)\end{aligned}$$

It is easy to prove that $agn\ 0$ is the bottom of E, using the fact that $1_\oplus$ is the bottom of c, and that $(E, \sqsubseteq_E)$ is a join semilattice.

5.2 Finding function K

In this subsection we derive a definition of function K satisfying (17), i.e., we construct a function K such that

$$agn\ (n{+}1) \quad = \quad K\ (agn\ n)$$

In the next subsection we show that K is monotonic with respect to $\sqsubseteq_E$, which allows us to conclude that $agn\ \infty$ is the least fixed point of function K, i.e.,

$$agn\ \infty \quad = \quad \mu K$$

We calculate as follows for $agn\ (n{+}1)$. The goal is to express $agn\ (n{+}1)$ in terms of $agn\ n$.

$$\begin{aligned}&agn\ (n{+}1)\\ =\ &\quad \text{definition of } agn\\ &(id \vartriangle (af \cdot gh\ (n{+}1) \cdot N)){*}\ (nts\ g)\end{aligned}$$

We proceed the calculation with the subexpression $af \cdot gh\ (n{+}1) \cdot N$. Suppose we can find a function J such that

$$af \cdot gh\ (n{+}1) \cdot N \quad = \quad J\ (agn\ n) \qquad (18)$$

then we have

$$agn\ (n{+}1) \quad = \quad (id \vartriangle (J\ (agn\ n))){*}\ (nts\ g)$$

and it follows by abstracting from $agn\ n$ in the right-hand side of this equation that a function K satisfying (17) is defined by

$$K\ x \quad = \quad (id \vartriangle J\ x){*}\ (nts\ g) \qquad (19)$$

It remains to find a function J such that equation (18) is satisfied.

Function J satisfying equation (18) is obtained by manipulating the expression $af\ (gh\ (n{+}1)\ (N\ nt))$, where nt is an element of $nts\ g$. Abbreviate the right-hand argument of $+\!\!+$ in definition (15) of $gh\ (n{+}1)$ to rh, that is

$$\begin{aligned}rh \quad = \quad &[Node\ nt\ c \mid rhs \leftarrow rhss\ g\ nt,\\ &\qquad c \leftarrow cp\ ((gh\ n){*}\ rhs)]\end{aligned}$$

Using this abbreviation we calculate as follows for $af\ (gh\ (n{+}1)\ (N\ nt))$.

$$\begin{aligned}&af\ (gh\ (n{+}1)\ (N\ nt))\\ =\ &\quad \text{definition of } gh\\ &af\ (gh\ n\ (N\ nt) +\!\!+ rh)\\ =\ &\quad \text{definition of } af\\ &(combine \cdot properties)\ (gh\ n\ (N\ nt) +\!\!+ rh)\\ =\ &\quad \text{definition of } combine \text{ and } properties\\ &(\oplus/ \cdot p{*})\ (gh\ n\ (N\ nt) +\!\!+ rh)\\ =\ &\quad (6)\\ &\oplus/\ (p{*}\ (gh\ n\ (N\ nt)) +\!\!+ p{*}\ rh)\\ =\ &\quad (11)\\ &(\oplus/ \cdot p{*})\ (gh\ n\ (N\ nt)) \oplus (\oplus/ \cdot p{*})\ rh\\ =\ &\quad af,\ combine, \text{ and } properties\\ &af\ (gh\ n\ (N\ nt)) \oplus af\ rh\end{aligned}$$

We express the arguments of operator $\oplus$ in the last expression above in terms of $agn\ n$ separately. For $agn\ n$ we have

$$agn\ n \quad = \quad (id \vartriangle (af \cdot gh\ n \cdot N)){*}\ (nts\ g)$$

and it follows that if we define function r by

$$r\ x\ a \quad = \quad at\ x\ a$$

where function at is defined by

$$at\ x\ a \quad = \quad head\ [y \mid (a, y) \leftarrow x]$$

then

$$af \cdot gh\ n \cdot N \quad = \quad r\ (agn\ n) \qquad (20)$$

This equation is used to express the left-hand argument of operator $\oplus$ in terms of $agn\ n$. It remains to express the right-hand argument of operator $\oplus$ in terms of $agn\ n$. We calculate as follows for $af\ rh$

$$\begin{array}{ll}
& af\ rh \\
= & \quad \text{definition of } rh \\
& af\ [Node\ nt\ c \mid rhs \leftarrow rhss\ g\ nt, \\
& \qquad\qquad\qquad c \leftarrow cp\ ((gh\ n){*}\ rhs)] \\
= & \quad \text{calculation from Section 4} \\
& (af \cdot (Node\ nt){*} \cdot +\!\!\!+/ \cdot cp{*} \cdot (gh\ n){*}{*}) \\
& (rhss\ g\ nt)
\end{array}$$

If we can push af to the right within the map $(gh\ n){*}{*}$ in the composition of functions of the last expression in the above calculation, then we can use equation (20) again to obtain an expression of the desired form. Aiming at pushing af to the right then, we proceed with the composition of functions $af \cdot (Node\ nt){*} \cdot +\!\!\!+/ \cdot cp{*} \cdot (gh\ n){*}{*}$. Abbreviate function $Node\ nt$ to mt.

$$\begin{array}{ll}
& af \cdot mt{*} \cdot +\!\!\!+/ \cdot cp{*} \cdot (gh\ n){*}{*} \\
= & \quad \text{definition of } af \\
& \oplus/ \cdot p{*} \cdot mt{*} \cdot +\!\!\!+/ \cdot cp{*} \cdot (gh\ n){*}{*} \\
= & \quad \text{map-distributivity (7)} \\
& \oplus/ \cdot (p \cdot mt){*} \cdot +\!\!\!+/ \cdot cp{*} \cdot (gh\ n){*}{*} \\
= & \quad \text{equation (12)} \\
& \oplus/ \cdot +\!\!\!+/ \cdot (p \cdot mt){*}{*} \cdot cp{*} \cdot (gh\ n){*}{*} \\
= & \quad \text{equation (13)} \\
& \oplus/ \cdot \oplus/{*} \cdot (p \cdot mt){*}{*} \cdot cp{*} \cdot (gh\ n){*}{*} \\
= & \quad \text{map-distributivity (7)} \\
& \oplus/ \cdot (\oplus/ \cdot (p \cdot mt){*} \cdot cp \cdot (gh\ n){*}){*}
\end{array}$$

At this point of the calculation we assume that there exists a function pn such that

$$p\ (Node\ nt\ x) \ = \ pn\ nt\ ((top \vartriangle p){*}\ x) \quad (21)$$

This condition is not unreasonable: for all *Rose-Tree* catamorphisms there exists such a function pn. We proceed the calculation with the expression within the map in the last expression of the above calculation.

$$\begin{array}{ll}
& \oplus/ \cdot (p \cdot mt){*} \cdot cp \cdot (gh\ n){*} \\
= & \quad \text{assumption (21)} \\
& \oplus/ \cdot (pn\ nt \cdot (top \vartriangle p){*}){*} \cdot cp \cdot (gh\ n){*} \\
= & \quad \text{map-distributivity (7)} \\
& \oplus/ \cdot (pn\ nt){*} \cdot (top \vartriangle p){*}{*} \cdot cp \cdot (gh\ n){*} \\
= & \quad (14) \\
& \oplus/ \cdot (pn\ nt){*} \cdot cp \cdot (top \vartriangle p){*}{*} \cdot (gh\ n){*} \\
= & \quad \text{map-distributivity (7)} \\
& \oplus/ \cdot (pn\ nt){*} \cdot cp \cdot ((top \vartriangle p){*} \cdot (gh\ n)){*} \\
= & \quad \text{introduction of function } zri \text{ below} \\
& \oplus/ \cdot (pn\ nt){*} \cdot cp \cdot zri{*} \cdot (id \vartriangle (p{*} \cdot gh\ n)){*} \\
= & \quad \textbf{assume} \text{ equation (22) below} \\
& H\ nt \cdot (id \times \oplus/){*} \cdot (id \vartriangle (p{*} \cdot gh\ n)){*} \\
= & \quad \text{map-distributivity (7), (5)} \\
& H\ nt \cdot (id \vartriangle (\oplus/ \cdot p{*} \cdot gh\ n)){*} \\
= & \quad \text{introduction of } r' \text{ below} \\
& H\ nt \cdot (r'\ (agn\ n)){*}
\end{array}$$

In this calculation we have assumed the existence of three functions: zri, H, and r' such that a number of properties is satisfied. Function zri is defined by

$$zri \quad = \quad \Upsilon \cdot repeat \times id$$

where function $repeat$ takes an element a, and returns an infinite list of a's. We omit the proof of the fact that function zri satisfies the following equality.

$$zri \cdot id \vartriangle (p{*} \cdot gh\ n) \ = \ (top \vartriangle p){*} \cdot (gh\ n)$$

Furthermore, we have assumed the existence of a function H such that the following equality is satisfied.

$$\begin{array}{ll}
& \oplus/ \cdot (pn\ nt){*} \cdot cp \cdot zri{*} \\
= & \\
& H\ nt \cdot (id \times \oplus/){*}
\end{array} \qquad (22)$$

Finally, function r' is defined by

$$\begin{array}{lcl}
r'\ x\ (N\ a) & = & (N\ a, r\ x\ a) \\
r'\ x\ (T\ b) & = & (T\ b, pl\ b)
\end{array}$$

The above derivation shows that if there exist a function H satisfying (22), then there exists a function J satisfying equation (18). Function J is defined by

$$\begin{array}{ll} & J\ x\ a \\ = & \qquad\qquad\qquad\qquad (23) \\ & (r\ x\ a)\ \oplus \\ & ((\oplus/ \cdot (H\ a \cdot (r'\ x)*)*)\ (rhss\ g\ a)) \end{array}$$

It remains to prove that there exists a function H such that equation (22) is satisfied, and that function K defined in equation (19) is monotonic. The latter condition is discussed in the following subsection, and the former condition in the subsection thereafter.

We give an operational interpretation of the functions we have derived. Given a grammar g and a CPO $(E, \sqsubseteq_E)$, we compute the least fixed point of function K, starting with $K\ \bot$, where $\bot$ is the bottom of E, and repeatedly applying K until we find a value x such that $K\ x = x$. Function K applies function J to all nonterminals of g. Function J takes the old value of K and a nonterminal nt, and returns the new value for nt by applying the function $H\ nt \cdot (r'\ x)*$ to all right-hand sides of the productions of nonterminal nt. The results are combined by taking the join $\oplus/$ of the values thus obtained, and, finally, by joining the result with the old value for nt.

5.3 Function K is monotonic

In order to guarantee the existence of the least fixed point of function $K : E \rightarrow E$ defined by

$$K\ x \quad = \quad (id \vartriangle (J\ x))*\ (nts\ g)$$

where function J is defined in equation (23), we have to show that function K is monotonic, i.e., for a, $a' \in E$, K has to satisfy

$$a \sqsubseteq_E a' \quad \Rightarrow \quad K\ a \sqsubseteq_E K\ a'$$

It is easily verified that function K is monotonic, provided function J is monotonic on E in its first argument, that is, provided

$$a \sqsubseteq_E a' \quad \Rightarrow \quad J\ a\ nt \sqsubseteq_E J\ a'\ nt$$

Function J is monotonic in a, provided operator $\oplus$ is monotonic in both its arguments, and function H satisfying equation (22) is monotonic in its second argument. These are the last conditions we impose upon the components of a bottom-up grammar analysis problem. An example of an operator $\oplus$ that is monotonic in both its arguments is the operator join $\sqcup_c$ of the semilattice c by means of which the semilattice E is defined. In the examples of the following subsection and the program in subsection 5.6, operator $\oplus$ will be the join of a join semilattice, which renders the verification of monotonicity of $\oplus$ trivial. Since K is monotonic and the domain of K is finite, the least fixed point of K can be found in finite time.

5.4 The conditions

In the previous subsections we have derived a function K by means of which a bottom-up grammar analysis problem can be solved. In the derivation we have imposed a number of conditions upon the components of the grammar analysis problems. This subsection discusses these conditions.

The first condition we imposed upon bottom-up grammar analysis problems is the following. We suppose there exists a join semilattice $(c, \sqsubseteq_c)$ such that $1_\oplus$ is the bottom of c, and $\oplus$ is the join $\sqcup_c$ of c.

For the second condition we suppose that there exists a monotonic function H, such that the following equality holds.

$$\begin{array}{ll} & \oplus/ \cdot (pn\ nt)* \cdot cp \cdot zri* \\ = & \\ & H\ nt \cdot (id \times \oplus/)* \end{array}$$

There exists a trivial but rather useless monotonic function H such that the above equality is satisfied:

$$\begin{array}{ll} & \oplus/ \cdot (pn\ nt)* \cdot cp \cdot zri* \\ = & \quad (1) \\ & \oplus/ \cdot (pn\ nt)* \cdot cp \cdot zri* \cdot (exl \cdot id \vartriangle \oplus/)* \\ = & \quad \text{map-distributivity (7)} \\ & \oplus/ \cdot (pn\ nt)* \cdot cp \cdot zri* \cdot exl* \cdot (id \vartriangle \oplus/)* \\ = & \quad \text{definition of } H \text{ below} \\ & H\ nt \cdot (id \vartriangle \oplus/)* \end{array}$$

where function H is defined by

$$\begin{array}{ll} & H\ nt \\ = & \\ & \oplus/ \cdot (pn\ nt)* \cdot cp \cdot zri* \cdot exl* \end{array}$$

Function H recomputes the required information from scratch instead of using the available information, and function H is therefore highly inefficient. However, it follows from this definition that the only condition that has to be satisfied in order to solve a bottom-up grammar analysis problem is the first condition given above. To obtain a practical solution for a bottom-up grammar analysis problem we discuss a special case in which we can find a monotonic function H that can be implemented as an efficient program.

Suppose the property function p is a catamorphism on *Rosetree*. Then we have for function $pn\ nt$ satisfying assumption (21)

$$pn\ nt \quad = \quad qn\ nt \cdot exr* \qquad (24)$$

where function qn is the function of the *Rosetree* catamorphism for p. For the left-hand expression of equation (22) we now calculate as follows.

$$\begin{array}{ll}
 & \oplus/ \cdot (pn\ nt)* \cdot cp \cdot zri* \\
= & \quad (24) \\
 & \oplus/ \cdot (qn\ nt \cdot exr*)* \cdot cp \cdot zri* \\
= & \quad \text{map distributivity (7)} \\
 & \oplus/ \cdot (qn\ nt)* \cdot exr** \cdot cp \cdot zri* \\
= & \quad \text{(14), map distributivity (7)} \\
 & \oplus/ \cdot (qn\ nt)* \cdot cp \cdot (exr* \cdot zri)* \\
= & \quad \text{product/zip calculation (omitted)} \\
 & \oplus/ \cdot (qn\ nt)* \cdot cp \cdot exr* \\
= & \quad \textbf{assume} \text{ equation (26) below} \\
 & qn\ nt \cdot (\oplus/)* \cdot exr* \\
= & \quad \text{map distributivity (7), (3), (7)} \\
 & qn\ nt \cdot exr* \cdot (id \times \oplus/)*
\end{array}$$

It follows that if we assume that there exists a function qn such that

$$\begin{array}{lcll}
p \cdot (Node\ nt) & = & qn\ nt \cdot p* & (25) \\
\oplus/ \cdot (qn\ nt)* \cdot cp & = & qn\ nt \cdot (\oplus/)* & (26)
\end{array}$$

then function H can be defined by

$$H\ nt \quad = \quad qn\ nt \cdot exr*$$

The second assumption is still rather unwieldy, and can be simplified. To obtain a simpler condition we apply the theory for cp developed in [4]. For that purpose, we first assume that function $qn\ nt$ is a reduction, that is, there exists an operator $\otimes$ with unit u such that

$$qn\ nt \quad = \quad \otimes/$$

Now we apply a theorem from [4], which states that (26) holds, provided the sections $(a\otimes)$ and $(\otimes a)$ distribute over operator $\oplus$.

$$\begin{array}{rcl}
a \otimes (b \oplus c) & = & (a \otimes b) \oplus (a \otimes c) \\
(a \oplus b) \otimes c & = & (a \otimes c) \oplus (b \otimes c)
\end{array}$$

and provided for all y, $1_\oplus \otimes y = y \otimes 1_\oplus = 1_\oplus$. Function $H\ nt$ is monotonic provided function $qn\ nt$ is monotonic, and function $qn\ nt$ is monotonic provided operator $\otimes$ is monotonic in both arguments.

5.5 Examples

This section shows how we apply the theory derived in the previous section to the examples of bottom-up grammar analysis problems given in Section 4. The algorithm derived in the previous section can be used to solve a bottom-up grammar analysis problem provided the components of the grammar analysis problem satisfy the conditions given in the previous section.

EMPTY
We verify the conditions the components of the definition of the bottom-up grammar analysis problem EMPTY have to satisfy.

First, the join semilattice $(c, \sqsubseteq_c)$ upon which the join semilattice $(E, \sqsubseteq_E)$ is built is the join semilattice of booleans, where c is the set $\{true, false\}$, the relation $\sqsubseteq_c$ is defined by $false \sqsubseteq_c true$, $false$ is the bottom of c, and the join $\sqcup_c$ is the operator $\vee$. Clearly, $\vee$ is associative, and $false$ is the unit of $\vee$.

For the second assumption, we have to construct a function H such that equation (22) holds. To obtain a definition of function H that can be implemented as an efficient program, we verify the conditions listed in the previous subsection. We have to show that function p defined by

$$p \quad = \quad ([\,] =) \cdot sen$$

is a *Rosetree* catamorphism, i.e., there should exist a function qn such that

$$p\ (Node\ nt\ x) \quad = \quad qn\ nt\ (p*\ x)$$

A definition of function qn is obtained as follows.

$$\begin{array}{ll}
 & p\ (Node\ nt\ x) \\
= & \quad \text{definition of } h \\
 & (([\,] =) \cdot sen \cdot Node\ nt)\ x \\
= & \quad \text{definition of } sen
\end{array}$$

$$((\,[\,] =) \cdot +\!\!+/ \cdot sen*)\ x$$
$$=\quad ([\,] =) \cdot +\!\!+/ = and \cdot ([\,] =)*,\ (7)$$
$$(and \cdot (([\,] =) \cdot sen)*)\ x$$
$$=\quad \text{definition of } qn \text{ (see below), and } p$$
$$qn\ nt\ (p*\ x)$$

Function qn is defined by

$$qn\ nt\ x \quad = \quad and\ x$$

Furthermore, we have to show that $\wedge$, the operator of the reduction for *and*, distributes over $\vee$, that is,

$$\begin{aligned} a \wedge (b \vee c) &= (a \wedge b) \vee (a \wedge c) \\ (a \vee b) \wedge c &= (a \wedge c) \vee (b \wedge c) \end{aligned}$$

and that *false* is a zero of $\wedge$. These equalities hold for *false*, $\vee$ and $\wedge$. Finally, we have to show that $\wedge$ is monotonic in both arguments. This requirement is satisfied too.

FIRST
We verify the conditions the components of the definition of the bottom-up grammar analysis problem FIRST have to satisfy.

First the join semilattice $(c, \sqsubseteq_c)$ upon which the join semilattice $(E, \sqsubseteq_E)$ is built is the join semilattice of terminals, where c is the set of terminals, the relation $\sqsubseteq_c$ is the subset relation, $[\,]$ is the bottom of c, and the join $\sqcup_c$ is set union, or $nub \cdot +\!\!+$. Clearly, set union is associative, and $[\,]$ is the unit of set union.

For the second assumption, we have to construct a function H that can be implemented as an efficient program, such that equation (22) holds. The condition (26) given in the previous subsection does not hold for function p defined by

$$p \quad = \quad take\ 1 \cdot sen$$

It is not difficult to find a *Rosetree* catamorphism for p, so (25) is satisfied, but the second requirement (26) does not hold. It follows that we have to find another way to construct function H. Function H is defined by

$$H\ nt \quad = \quad foldr\ t\ 1_{\oplus}$$

where function t is defined as follows. If the current symbol in the right-hand side of a production is a terminal, then the symbols that can appear as the first symbol of a string are the symbols found until then, and no more.

$$t\ (T\ b, y)\ x \quad = \quad y$$

If the current symbol in the right-hand side of a production is a nonterminal $N\ a$, then we distinguish two cases depending on whether or not $N\ a$ can derive the empty string. If $N\ a$ can derive the empty string, then the symbols that can appear as the first symbol of a string are the symbols found until then together with the first symbols of the remaining part of the production. If $N\ a$ cannot derive the empty string then the symbols that can appear as the first symbol of a string are the symbols found until then, and no more.

$$t\ (N\ a, y)\ x$$
$$= \begin{cases} nub\ (y +\!\!+ x) & \text{if } at\ emptiesa \\ y & \text{otherwise} \end{cases}$$

We can prove equation (22) for function H thus defined by induction to the structure of lists: apply both sides to $[\,]$ and $[a] +\!\!+ x$, and show that the resulting expressions have the same recursive structure. The proof of the fact that H is monotonic is easy and omitted.

5.6 Implementation

The definitions given in the previous subsections are translated into Gofer as follows. Some rather obvious alterations of these functions increase the efficiency of the program. These alterations are discussed after the following program.

```
ag g pn pl
  = lfp
    k
    (map (split id (\x -> bottom)) nt's)
    where
    nt's = nts g
    k x = map (split id (j x)) nt's
    j x nt = (r x nt)
             'join'
             ((lub
               .map (pn nt . map (r' x))
               .rhss g)
               nt
             )
    r x nt =  at x nt
    r' x (N a) = (N a, r x a)
    r' x (T b) = (T b, pl b)
```

We discuss two of the possibly many ways in which a more efficient program can be obtained.

Function `lfp` applies a function `f` to an argument `x` until it reaches a fixed point. In order to determine whether an argument is a fixed point, `f x` is compared with `x`. In the case of grammar analysis problems, the first components of the elements of the grammar analysis problems are always the nonterminals of the given grammar. It follows that the first components of the elements of `f x` and `x` are always equal, and that equality depends just on the second components of the problems. The first condition of function `lfp` may be replaced by `map snd x = map snd (f x)`.

Another way to improve the performance of the program is to replace `map (j x) nt's` by `map (j x) g`, and to replace the definition of function `j` by

```
j x (nt,pnt) = join
               (r x nt)
               ((lub
                .map (pn nt . map (r' x)))
                pnt
               )
```

6 Conclusions

This paper discusses bottom-up grammar analysis problems. We give a very general specification of bottom-up grammar analysis problems, and from this specification we derive, by means of program transformation applying laws to the components of the intermediate expressions, an algorithm for performing bottom-up grammar analysis. The driving force in the derivation of the algorithm is the construction of a fixed point. To obtain such a fixed point a number of conditions have to be imposed upon the components of the bottom-up grammar analysis problem. Thus we derive both the algorithm and the conditions under which the fixed point exists in one go. The derivation is an example of a derivation of a real-world program, which would have been difficult to obtain without a derivation. The research reported on in this paper is still in progress: in the next version we want to split the calculation in two parts. The first part of the derivation assumes that the function that computes the property of a parse tree is a *Rosetree* catamorphism and the second part of the derivation adds, if necessary, the extra information (for example in the case of *firsts*, where we use information about the *empties*). This simplifies the derivation. Future research will be directed towards the derivation of an algorithm for top-down grammar analysis.

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

[3] J.H. Fasel, P. Hudak, S. Peyton Jones, and P. Wadler. Sigplan Notices Special Issue on the Functional Programming Language Haskell. *ACM SIGPLAN notices*, 27(5), 1992.

[4] J. Jeuring. *Theories for Algorithm Calculation*. PhD thesis, Utrecht University, 1993.

[5] M.P. Jones. Introduction to Gofer 2.20. Programming Research Group, Oxford University, 1992.

[6] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[7] 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.

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

[9] 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.

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

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